THE SUPER-USER'S BBC MICRO BOOK

Brian James and Graham Keeler

The Super-User's BBC Micro Book

The Super-User's BBC Micro Book

Brian James and Graham Keeler
University of Salford

with contributions from

Herbert Dunderdale
Cedric Owen
Gerald Tomlinson

ADDISON-WESLEY PUBLISHING COMPANY

Wokingham, England · Reading, Massachusetts · Menlo Park, California
Don Mills, Ontario · Amsterdam · Sydney · Singapore · Tokyo
Mexico City · Bogota · Santiago · San Juan

© 1985 Addison-Wesley Publishers Limited
© 1985 Addison-Wesley Publishing Company, Inc

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the publisher.

The programs presented in this book have been included for their instructional value. They have been tested with care but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs.

Cover design by Jonathan Doney.
Photoset direct from the authors' text by Quorum Technical Services Ltd.
Printed in Great Britain by The Bath Press.

British Library Cataloguing in Publication Data
James, Brian
The super-user's BBC micro book.
1. BBC Microcomputer—Programming
I. Title II. Keeler, Graham
001.64'2 QA76.8.B35

ISBN 0-201-15435-8

Library of Congress Cataloging in Publication Data
James, Brian.
The Super-User's BBC micro book.

Includes index.
1. BBC Microcomputer—Programming. 2. BASIC (Computer program language) I. Keeler, Graham, 1943– II. Title.
QA76.8.B35J35 1985 005.2'65 85-20148
ISBN 0-201-15435-8 (pbk.)

ABCDEF898765

To wives and daughters

Econet is a trademark of Acorn Computers Limited.
Wordwise and *Disc Doctor* are trademarks of Computer Concepts.
Intel 8271 is a trademark of the Intel Corporation.
MC68B54 is a trademark of Motorola Inc.

CONTENTS

Preface		xi
Chapter 1	**Introduction**	1
1.1	The BBC microcomputer	1
1.2	About this book	2
1.3	Using the book	3
Chapter 2	**Programming statements and structures**	5
2.1	Standard BASIC commands	13
2.2	Structured commands available on the BBC microcomputer	17
Chapter 3	**Structured programming facilities**	22
3.1	Introduction	22
3.2	Functions	23
3.3	Local parameters	28
3.4	Procedures (DEF PROC)	30
3.5	Other program structures	32
3.6	Recursive use of procedures and functions	34
Chapter 4	**Strings**	39
4.1	String variables	39
4.2	Printing with strings	40
4.3	Comparison of string variables	42
4.4	String manipulation functions	43
4.5	String arrays	50
4.6	Further string handling	52
Chapter 5	**Mathematical functions**	56
5.1	Standard functions	56
5.2	Random numbers	58
5.3	Logical expressions	62

Chapter 6 Graphics 69

6.1 Introduction 69
6.2 Plotting lines and curves 74
6.3 The PLOT and VDU graphics commands 81
6.4 Special graphics effects 86

Chapter 7 Programming errors 94

7.1 Syntax errors 95
7.2 Semantic errors 97
7.3 Execution errors 99
7.4 Rerunning the program 100
7.5 Error trapping 102

Chapter 8 Use of the disc filing system 106

8.1 Introduction 106
8.2 Structure of information on a disc 109
8.3 Formatting and verifying 112
8.4 Other DFS commands 115
8.5 Turnkey systems 123

Chapter 9 File handling 125

9.1 Types of file 125
9.2 Handling data files 127
9.3 Further aspects of data files 132
9.4 Random access data files 135
9.5 Single byte access to a data file 138
9.6 A general program to read or write a sequential data file 140
9.7 ASCII files 143

Chapter 10 Direct memory and system access 151

10.1 Computer memory 151
10.2 Organization of memory in the BBC microcomputer 153
10.3 The memory pointers 155
10.4 Memory access 159
10.5 A MEMZAP and DISCZAP program 162
10.6 The way BASIC programs are stored 168
10.7 Handling the filing and operating systems from within programs 174
10.8 Using machine code programs 178

Chapter 11 Programming techniques 187

11.1 'User-friendly' programming 187
11.2 Input facilities of BBC BASIC 188
11.3 Validation 193
11.4 Display layout 197
11.5 Formatting numerical output 201
11.6 General presentation 206
11.7 Menu driven programs 210

Appendix A	**Special features of the BBC microcomputer**	214
Appendix B	**Selected *FX commands**	219
Appendix C	**Summary of the VDU commands**	224
Appendix D	**Bugs and unexpected effects in BBC BASIC**	226
Appendix E	**Differences between BASIC I and BASIC II**	230
Appendix F	**The resident variable @%**	232
Appendix G	**Pseudo-variables in BBC BASIC**	236
Appendix H	**ASCII codes**	237
Appendix I	**INKEY codes**	239
Appendix J	**Error codes**	241
Appendix K	**Printers**	243
Appendix L	**The Econet filing system**	247
Appendix M	**Using the DISCZAP program**	262
Appendix N	**Answers to exercises**	266
Appendix O	**The supplementary disc and tape**	284
Index		285

PREFACE

The BBC microcomputer is in many ways outstanding for a small microcomputer in the wide range of facilities built into the basic machine and in the power and range of its language and operating system. Indeed, BBC BASIC is so much more extensive and better structured than the versions of BASIC found on most other small microcomputers that it almost needs to be considered as a separate language, of which standard BASIC is but a subset.

The aim of this book is to introduce readers to this treasure house of possibilities, although even so we have not been able fully to cover all aspects. In graphics, for instance, although we have devoted a full chapter to the topic we have but scratched the surface. A full treatment of this particular BBC microcomputer feature would take a complete book in itself – and indeed there are already several books devoted entirely to graphics and sound. Our endeavour instead has been to try and explore the full breadth of the system, and include areas which have been less widely discussed.

The book is not aimed at the complete beginner, although we have very briefly reviewed the fundamental aspects of the BBC computer and BASIC in the first two chapters. Rather, it is designed for the user who has made a start but who finds the need for help in order to progress further. It would also be ideal for users who are familiar with other microcomputers but would like a conversion course to the BBC microcomputer.

The book is based on a short course called 'Getting more from your BBC microcomputer' run jointly by the Department of Pure and Applied Physics and the Department of Electronic and Electrical Engineering at the University of Salford. Each participant is provided with an extensive manual including all the exercises for the course, and this manual was written jointly by us and our fellow contributors.

It is this original manual that we have revised and extended to produce this book, and we have included in this book revisions and adaptations in the light of our experience on the course, concentrating extra discussion and explanation on those aspects that participants have found interesting or difficult.

Following the style of our course, the exercises form an integral part of this book, and are included in the body of the text rather than at the end of

each chapter. The ideal way to use the book is with a BBC microcomputer to hand, trying out the exercises as they are reached. The book also includes a large number of example programs. For those who would like to save the time taken to type in all these programs, they are also available on a supplementary disc or tape.

We would like to express our thanks to our colleagues, Bert Dunderdale, Cedric Owen and Gerry Tomlinson for their contributions to the original manual. We would also thank our wives, Wendy James and Marion Keeler, for their help with the typing of the text on the *Wordwise* word processor, and Wendy James for her help in reading the draft manuscript and preparing some of the programs.

Chapter 1

INTRODUCTION

1.1 THE BBC MICROCOMPUTER

The BBC microcomputer was initially produced in two versions – the Model A and the Model B. The Model B was very much more popular, and so the Model A is no longer readily available. Much more recently a third model has been launched, the Model B+. This book is written primarily for the Model B and B+ which are almost entirely compatible as far as software is concerned; though most of the contents will also apply to the Model A, there will be certain points, such as use of Modes 0 to 3, and the description of the Disc Filing System (DFS), which do not apply. Model A users should therefore be aware that there may be occasions when the comments made do not apply to their computer.

The computer's software has been revised during the life of the machine. This software comprises two parts: the BASIC interpreter and the Machine Operating System (MOS). The BASIC interpreter has appeared in two versions. The original version is now usually referred to as BASIC I, and the revised version as BASIC II. The changes are mostly very minor, except for certain changes to data file handling, which are explained in Chapter 9. The differences between the two are summarized in Appendix E.

The original MOS was called OS 0.1. This was replaced by OS 1.0, which was very quickly superseded by OS 1.2. The differences between the last two are minor, but OS 0.1 was a very restricted MOS. You can find out which version your computer has by typing *FX 0. If it is OS 0.1, you should consider having it updated. This book assumes a version of at least 1.0, and some features may not be correct for 0.1.

The computer Machine Operating System (MOS) is designed to work with a variety of different *filing systems*; that is, methods of storing and retrieving programs, data or other information. The original, built-in filing system is the cassette filing system, and many individual users will be using this. It suffers from a number of shortcomings. It is very slow, and being a serial recording system certain operations such as random access of data files and chaining back and forth between files are not easily possible. There is also a practical problem in that cassette recorders are not designed for digital recording, and vary widely in quality and performance. Moreover,

they are not completely reliable, particularly for interchange amongst different recorders.

The Disc Filing System (DFS) is the other main system for individual users. Disc drives are precision instruments providing a very rapid form of data access, and the BBC DFS filing system is particularly fast in this respect, compared with other disc systems on microcomputers. The DFS provides a more comprehensive filing system than the cassette filing system, and includes features such as random access and the possibility of having several data files open simultaneously. A complete chapter of this book is devoted to the DFS, and certain aspects of the chapter on file handling also apply only to the use of the DFS.

A third filing system available for the BBC microcomputer is the Econet filing system. This is designed for users of groups of computers, so that they can share facilities such as disc storage and printing. One computer has to be dedicated to the Econet system, as *file server*, running a file server program which manages the whole network. There are two main versions of the file server, and the systems that they run are denoted as level 1 and level 2. Level 1 provides a rather primitive filing system, allowing only the transfer of complete files. Level 2 is a much more sophisticated system, allowing nearly all the facilities of the DFS, other than a few commands mainly concerned with ASCII files, such as *BUILD and *TYPE. None of these omissions is serious, and could quite easily be replaced by user-written programs. Level 2 does, however, require a computer fitted with a 6502 second processor, and as a minimum a double-sided 80 track disc drive. Appendix L discusses the Econet filing system in more detail.

1.2 ABOUT THIS BOOK

Although this book starts with a brief review of simple BASIC commands, it does not set out to teach the fundamentals of BASIC. There are many other books already available which do this. Instead, the aim here in this book is to introduce the user to many of the more advanced and sophisticated features available on the BBC microcomputer. Some of these are general features of BASIC, but others are available only on the BBC computer, which has a version of BASIC far superior to that available on most other microcomputers.

This book is therefore suitable for readers who have gained familiarity with the BBC computer, but would like to explore it in more depth and learn the full range of its capabilities. It would also be suitable for readers familiar with BASIC on a different computer, who would like to learn and become familiar with its use on the BBC computer in particular.

For users not already familiar with all the special features of the BBC microcomputer, Appendix A contains a brief summary of those features which are not dealt with in detail elsewhere in this book. It will be assumed throughout the remainder of the book that readers are familiar with these features.

Although the book can be read sequentially, it contains a wide variety of topics, and many readers will want to use it for chapters dealing with topics of

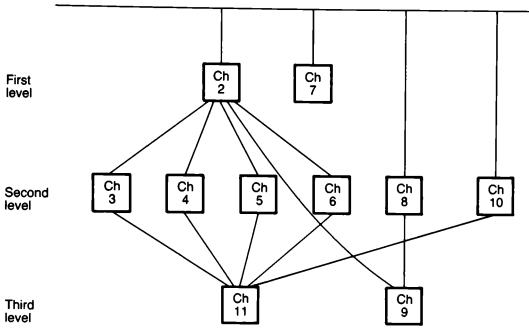


Figure 1.1 Links between chapters of the book.

particular interest to them. Readers interested in file handling on disc, for instance, might turn straight to Chapters 8 and 9.

There are links between some of the chapters, however; as far as possible, ideas needed in later chapters are introduced in earlier ones, although inevitably it is necessary in some cases to make use of techniques not fully explained until later. Figure 1.1 shows the major links between different chapters.

The book ends with a series of appendices giving tables, lists and a variety of detailed information, as well as answers to some of the exercises set throughout the book.

1.3 USING THE BOOK

The style of the book follows that of the short course on which it is based. It is written on the assumption that you can work at your computer as you read through the book. Thus exercises are not relegated to the end of the chapter, but form an integral part of the text. Although there is no reason why you should not read the book without recourse to a computer, you will probably learn more quickly, reinforce the ideas presented and understand the comments made more fully, if you can try things out at the computer as you go along.

A lot of the examples presented are in the form of short programs. It should be emphasized that these are, in the main, provided to illustrate a point or demonstrate a principle. They are deliberately kept as short and simple as possible, and do not include 'user-friendly' features such as input validation or error protection. Some programs which contain endless loops

do not even include a facility to quit! You will find it necessary to use <ESCAPE> to break out of them. The omission of the provision to quit the loop is not a mistake; the inclusion of escape options in every program would tend to obscure the points being made.

There are quite a number of programs of substantial length included in the book, and typing in all of these will be quite time-consuming. It is possible to purchase a disc which contains many of the longer programs listed in the book, as well as one or two which were too long to permit their inclusion here. Details of the disc and how to obtain it are given in Appendix O.

Note that here and throughout this book, angle brackets are used to mean that what is in the brackets should not be typed literally. Thus <ESCAPE> means the ESCAPE key, and <filename> means that an actual name of a file should be type in place of <filename>.

Chapter 2

PROGRAMMING STATEMENTS AND STRUCTURES

2.1 STANDARD BASIC COMMANDS

2.1.1 Variables and assignment statements

The BBC microcomputer allows the user a wide choice of names for the numerical variables that are used to identify the various parameters in a calculation. They may consist of upper and lower case letters, numbers, the pound sign and the underline character.

The important constraints are

1. They must begin with an upper or lower case letter. Starting a variable name with a number is not permitted. Every character in the name is significant, which means that TEMPERATURE1 is different from TEMPERATURE2.
2. It is not permissible to have spaces in variable names but the underline character can be used instead. Thus 'first number' is not acceptable but 'first_number' is in order.
3. Variable names must not contain punctuation marks such as , . ? ! or :.
4. Although BASIC keywords may be embedded in variable names (for example, CONCISE contains the BASIC keyword ON), they must not start with BASIC keywords, unless the keyword is in lower case letters, or partially in lower case letters. So 'today' and 'Today' are permitted but 'TODAY' is not because it starts with 'TO'.

It can also be convenient to use integer variables in a program, for counting purposes perhaps, and the addition of the percent sign, %, at the end of a variable name indicates an integer variable which will not suffer from the round-off errors associated with the binary arithmetic used by computers. The BBC microcomputer has a set of *resident integer variables*, A% to Z%, which are permanently allocated space in memory. Typing RUN or NEW does not destroy them or change their values, so they can be used to pass values from one program to the next. Their values will be lost if the computer

is switched off, but they will not be lost after CLEAR, <BREAK> or even <CTRL-BREAK>.

Apart from the resident integer variables, the first appearance of any numerical variable should be in the left hand side of an assignment statement. Thus to set the value of TYRE_PRESSURE in a program it must be used in a statement such as

```
LET TYRE_PRESSURE = 32.5
```

Thereafter TYRE_PRESSURE will have the value 32.5 and the statement

```
PRINT TYRE_PRESSURE
```

will yield the result

```
32.5
```

The statement

```
X=X+1
```

is an exception to the general rule that a variable must appear on the left hand side of an assignment statement before it appears in any BASIC statement in which its numeric value is used. In this special case the value of X on the right hand side is initialy taken to be zero.

An attempt to PRINT a variable before it has been assigned a value will give the error message 'No such variable'. If this occurs within a program the line number will also be given.

The BASIC keyword LET may be omitted in an assignment statement, although beginners are encouraged to use it for a while to become familiar with the concept of the use of the 'equal' sign, =, in an assignment statement to mean 'assign the the value on the right to the variable on the left'. This is opposed to the normal mathematical meaning 'is equal to' in a statement of equality such as 2+2=4. In BASIC the equal sign is used at various times to mean 'set equal to' and 'is equal to'. The second of these is used in tests for equality in IF statements. Both are used on the BBC microcomputer, the meaning being determined by the context. It is thus perfectly correct to have the statements

```
10 X=1
20 PRINT X
30 X=X+5
40 PRINT X
```

where line 10 sets the variable X to 1, and this is printed at line 20. At line 30, X is increased by the addition of 5 to the original value of X and is thus set to

6, which will be printed at line 40. In this program = has been used to mean 'set equal to'.

Lines 10 and 30 could equally well have been written as

```
10 LET X=1
30 LET X=X+5
```

which makes the meaning more obvious.

Note that if line 10 is removed from the above program, attempting to RUN it will produce the message

```
No such variable at line 20
```

The normal arithmetic operations of mathematics: addition, subtraction, division, multiplication, and raising to a power are represented in BASIC by + − / * and ^ respectively. The right hand side of an assignment statement may contain calculations involving any variables already given values and any sensible combination of these operations. For example

```
LET X=P*Q+R*S
LET D=B^2−4*A*C
LET Y=(A+B)/(C+D)
LET AV=(X+Y+Z)/3
```

The usual rules of precedence for evaluating these operations and the brackets apply, as follows

1. Brackets are evaluated (innermost first).
2. Raising to the power.
3. Multiplication and division.
4. Addition and subtraction.

Where operations of the same level of precedence occur together, the computer evaluates them from left to right. It is worth remembering this point if very large or very small numbers are involved in a calculation, as the following example will illustrate.

Example 2.1

```
10 BIG1=10E20
20 BIG2=20E20
30 BIG3=1E20
40 RESULT=BIG1*BIG2/BIG3
50 PRINT RESULT
```

It will be found that when this program is run the error message

```
Too big at line 40
```

will be obtained, since BIG1*BIG2 is 2×10^{42} and the computer cannot deal with numbers larger than 10^{38}. However, rewriting line 40 as

```
40 RESULT=BIG1/BIG3*BIG2
```

the calculation will be performed in the different order, giving the correct answer 2E22.

If the variables BIG1,BIG2,and BIG3 are set to 10E−22, 20E−20, and 1E−20 respectively, then with the first version of line 40 the value printed will be 0, and with the second, preferred form of line 40 the correct result 2E−20 is printed. There is no error message 'Too small'; instead the value is set to 0. But if the variable RESULT were used in other calculations in the same program after it had been set to zero, they would also give incorrect results and might even cause an error if a division by RESULT were attempted, when the error message

```
Division by zero
```

would be printed. This indicates the need for care in setting out calculations involving very small numbers as well as very large numbers.

2.1.2 GOTO statements

Computers normally execute statements one line at a time in the order in which they occur in the program. In BASIC the lines occur in the order in which they are numbered. However, it is often necessary to alter the order of execution in order to execute one block of the program several times, or to make the order depend on the values being processed during the program. Changing the order of execution is known as *branching* and the simplest way of branching is by means of the GOTO statement. The GOTO instruction alters the normal sequence and transfers execution to the line specified in the command. This line can be either before or after the line containing the GOTO statement.

Example 2.2

The following program prints out the squares of the integers.

```
10 M=0
20 M=M+1
30 PRINT M;" SQUARED = ";M^2
40 GOTO 20
50 END
```

In this example the computer goes into a never ending loop. The program can be stopped by pressing the <ESCAPE> key. It should be observed that any lines following line 40 would never be executed.

2.1.3 IF...THEN conditional statement

One way in which the previous program could be modified to cause the block from lines 20 to 40 to be executed a finite number of times, followed by

execution of any subsequent part of the program, is by means of the IF...THEN instruction. For example

```
IF M<N THEN GOTO 20
```

This statement allows the computer to take a decision. If the condition following the IF is TRUE, the computer executes the instruction following the THEN, otherwise the computer moves on to the next line. The previous exercise can now be modified by introducing line 5, and altering line 40 as follows

Example 2.3

```
 5 INPUT "NUMBER OF INTEGERS",N
10 M=0
20 M=M+1
30 PRINT M;" SQUARED = ";M^2
40 IF M<N THEN GOTO 20
50 END
```

The program now prints the squares of the first N integers, where N is input from the keyboard when the program is run. Each time the program passes through line 40 the test M<N is carried out. If the condition is TRUE then the program goes to line 20 next, but if the result of the test is FALSE, the program goes on to the next line, line 50.

The IF...THEN construction can be used with intructions other than GOTO. For example

```
IF X<0 THEN PRINT "NEGATIVE"
```

Note, also, that the GOTO can be omitted following the THEN, so that line 40 in the previous exercise could be written as

```
40 IF M<N THEN 20
```

On the BBC computer the THEN can be omitted from any IF...THEN construction. This is not standard BASIC.

String variables can be used in the conditional part of the IF...THEN statement, as in the following example.

Example 2.4

```
10 INPUT "WHAT IS THE CAPITAL OF FRANCE",A$
20 IF A$="PARIS" THEN 60
30 PRINT "WRONG, TRY AGAIN"
40 INPUT A$
50 GOTO 20
60 PRINT "CORRECT"
70 END
```

This program will run until the correct answer is given. In this example exact equality of the strings is necessary to satisfy the test. The greater than (>) and less than (<) signs can be used for conditional tests involving strings. The test A$<B$ is true if the contents of A$ come before the contents of B$ alphabetically.

Exercise 2.1

Modify the program above to permit up to two attempts, after which the correct answer is given.

On the BBC microcomputer several statements can be typed on the same line if they are separated by colons (:). In an IF...THEN statement, all the statements on the same line following the THEN will be obeyed if, and only if, the condition tested is true. For example,

```
20 IF A$="PARIS" THEN PRINT "CORRECT": GOTO 70
```

This feature can be very useful, but care must be taken if program lines are concatenated in order to save space, at a later date. Otherwise there is the chance that statements concatenated onto the end of a line containing a THEN may be inadvertantly skipped.

The above programs have used tests for 'less than' and 'equal to'. These are relational operators, and the full range is given in Table 2.1.

Table 2.1 The relational operators.

Symbol	*Meaning*
>	greater than
<	less than
=	equal to
>=	greater than or equal to
<=	less than or equal to
<>	not equal to

The order of the pairs of symbols must be as shown, and with no spaces between them. They can also be used to compare strings, as seen in the Example 2.4. The conditions in an IF statement can be combined with the Boolean operators AND, OR, NOT and EOR. This is fully discussed in the section on logical expressions in Chapter 5.

In the program above, the = sign is used to test for equality; that is, used to mean 'is equal to'. The results in language form would be YES (TRUE) or NO (FALSE), but as the computer is digital, the result is expressed as a number. On the BBC microcomputer −1 is used to indicate TRUE and 0 is used to indicate FALSE.

2.1.4 FOR...NEXT

A very widely used program construction is the FOR...NEXT loop, that is used if a sequence of processes is to be repeated (or iterated) a number of times. The syntax is

```
FOR J=<lower limit> TO <upper limit> STEP <increment>
...  }
...  }  other program statements
...  }
NEXT J
```

where <lower limit>, <upper limit> and <increment> may be numbers or numeric variables or numerical expressions. The STEP <increment> can be omitted, in which case a step size of +1 is assumed.

The FOR...NEXT construction works like this: starting with a FOR statement a variable (J in this case) is given the value of the lower limit. J may then be used for some calculation within the loop. Eventually the computer comes to a line with the statement NEXT (meaning next value of the variable please). This sends the computer back to the FOR statement, increases the variable by the amount of the increment, and checks that the upper limit has not been exceeded, after which the cycle repeats. If the upper limit has been exceeded, the loop is complete, and the program will continue with the statement following NEXT.

To impose any other change than the default increment of +1 on the variable, the FOR statement should finish with STEP <increment>. This can be any real number (positive or negative, integer or fractional) so that the count variable can be incremented or decremented by any amount. For example

```
FOR J=1 TO 5              J will take the values 1, 2, 3, 4, 5
FOR J=1 TO 10 STEP 2      J will take the values 1, 3, 5, 7, 9
FOR J=5 TO 1 STEP -1      J will take the values 5, 4, 3, 2, 1
FOR J=1 TO 2 STEP 0.25    J will take the values 1, 1.25, 1.5, 1.75, 2
```

A FOR...NEXT loop is always executed at least once in BBC BASIC. The test to see whether J is greater than the upper limit is not carried out until the loop has been executed once. If the 'increment' is negative, J will be decreased each time NEXT is reached, and the test will be whether J is less than the 'upper' limit or final value. In this case, of course, the final value must be less than the starting value for J. For example

```
FOR J=5 TO 1 STEP -1
```

would make J count correctly 5, 4, 3, 2, 1 but

```
FOR J=1 TO 5 STEP -1
```

would only execute the loop once with J=1. J would then be decreased to 0 and the loop would terminate because 0 is less than the final value of 5. This prevents the computer from getting into an infinite loop.

If a fractional value is used for the increment, it should be remembered that simple decimal fractions such as 0.1 are recurring fractions in binary and therefore cannot be represented exactly. This means that a FOR loop such as FOR J =1 TO 5 STEP 0.1 may yield a value of 4.9999999 rather than 5. It is preferable to use integer variables in FOR...NEXT loops when integers are intended.

The program in Example 2.5 below illustrates the use of a FOR...NEXT loop with an increment of 0.1. It will be seen that the two numbers printed are different by a factor of 10 until I=2.4. At that value round off errors affect which element of the A(I) array is printed. (Use <CTRL-N> to select paged mode for the display before running the program.)

Example 2.5

```
10 DIM A(50)
20 FOR I=1 TO 50
30 A(I)=I
40 NEXT
50 FOR I=1 TO 5 STEP 0.1
60 PRINT I,A(I*10)
70 NEXT
```

It is a useful exercise to change lines 20, 30, 50 and 60 as below

```
20 FOR I%=1 TO 50
30 A(I%)=I%
50 FOR I%=1 TO 50 STEP 1
60 PRINT I%/10,A(I%)
```

In this modified form the program will correctly print the numbers 10 to 50 alongside the numbers 1.0 to 5.0. The use of the resident integer variables A% to Z%, or integer variables, such as VAR%, overcomes any problem that is likely to occur due to round off error in cases where integer values are needed, such as for array subscripts.

The program in Example 2.6 is a summation program in which the sum of the ten prime numbers in the DATA statement is obtained.

Example 2.6

```
10 SUM=0: REM INITIALIZE SUM
20 FOR J%=1 TO 10: REM ITERATION TO FORM SUM
30 READ X: REM OBTAIN PRIME FROM DATA
40 SUM=SUM+X: REM ADD TO PREVIOUS SUMMATION
50 NEXT J%: REM LOOP BACK
60 PRINT "Sum of first 10 primes = ";SUM: REM PRESENT RESULT
70 END
80 DATA 1,2,3,5,7,11,13,17,19,23
```

Exercise 2.2

Write a program to calculate the sum of the first N whole numbers, where N is supplied in response to an INPUT statement by the user. This program should not use a DATA statement.

2.2 STRUCTURED COMMANDS AVAILABLE ON THE BBC MICROCOMPUTER

2.2.1 REPEAT...UNTIL statements

The BBC microcomputer has an alternative loop structure which is not normally available with BASIC. If a sequence of program statements is to be repeated until some condition is satisfied, rather than for a fixed number of times, then the REPEAT and UNTIL statements are appropriate. The sequence of statements is initialized by the REPEAT statement and ends with an UNTIL statement followed by a conditional expression. It is possible to have more than one UNTIL corresponding to the REPEAT provided that only one can possibly be executed. Thus we could have

```
200 IF FLAG=1 THEN UNTIL EPS>0.01 ELSE UNTIL EPS>0.0001
```

One common use of the REPEAT...UNTIL loop is to produce a delay during the operation of the program. This can be done with FOR...NEXT loops but better precision is obtained if the pseudo-variable TIME, available with the BBC microcomputer, is used. TIME is incremented by one every 0.01 second. A delay of 1.5 seconds will be obtained by use of the lines 40 and 50 in Example 2.7.

Example 2.7

```
10 REM Time delay of 1.5 second
20 PRINT "START OF DELAY"
30 START=TIME
40 REPEAT
50 UNTIL START+150>=TIME
60 PRINT "END OF DELAY"
```

An alternative is to change lines 30 and 50 to

```
30 TIME=0
50 UNTIL TIME>=150
```

The former is preferable in case the pseudo-variable TIME is being used for any other timing processes. Notice that in the UNTIL the conditional test is >=. It is written this way so that it is not necessary for the precise value to be satisfied at the instant the TIME variable is accessed. This might not occur due to the time taken to execute BASIC statements particularly if there were more lines between lines 40 and 50.

Exercise 2.3

Write a program to make the computer run a simple 'MASTERMIND' game in which the contestant has to guess the number selected by the computer in

the range 1 to 10. The program should use a REPEAT...UNTIL loop in its main section. (Use RND(10) to generate the number.)

Exercise 2.4

Extend the last program so that the number selected is in the range 1 to 1000 and the computer gives one of two clues: 'TOO LARGE' or 'TOO SMALL' until the correct number is guessed. (It should be possible to identify the number selected in 10 steps; if this is not achieved the program could be written to comment on the poor performance. Use RND(1000) to choose the number.)

2.2.2 IF...THEN...ELSE... structure

The IF...THEN statement is common to many implementations of BASIC. In BBC BASIC the statement is extended to include the ELSE condition. This construction is a valuable aid to clear, well structured programming in which the effect of statements in the program are immediately clear and it is not necessary to jump to different line numbers to follow the logic of a program. The IF...THEN...ELSE statement takes the form

```
IF <condition> THEN <statement(s)> ELSE <statement(s)>
```

The condition may be complex and contain any of the Boolean statements. For example IF (A>0) AND (A<10) THEN... is valid. The brackets are useful to clarify the order of operations. A simple example of the use of IF...THEN...ELSE is shown in Example 2.8.

Example 2.8

```
10 REPEAT
20 INPUT "A", A
30 IF (A>=0 AND A<=10) THEN PRINT "A in the range 0 to 10"
   ELSE PRINT "A out of range 0 to 10"
40 UNTIL A=999
```

The statements may also be complex within what can be contained in one line of BASIC. (Remember it is possible to compress a line on input by use of the abbreviations P. for PRINT etc.)

A single line of BASIC may contain more than one IF...THEN...ELSE statement. The individual assignments or statements following THEN or ELSE are separated by a colon in the usual way. The statements used may include calls to functions and procedures. This feature is particularly valuable for clear structured programming and avoids the need for GOTO and GOSUB with incomprehensible line numbers.

In multiple use of IF...THEN...ELSE in a single statement, where there are choices amongst more than two alternatives, it can only make clear logical sense if the order

```
IF...THEN...ELSE IF...THEN...ELSE...
```

is followed. If the sequence

```
IF...THEN IF...THEN...ELSE...ELSE...
```

were to be used it would no longer be clear which test is associated with which ELSE. Where a further IF test is required to follow a THEN, the test can be inverted by use of NOT so that the structure becomes

```
IF NOT <test1> THEN <statement(s)3> ELSE IF <test2> THEN
<statement(s)1> ELSE <statement(s)2>
```

Note that <statement(s)3> can be a null statement if appropriate, leaving the structure as

```
IF NOT <test1> THEN ELSE IF <test2> THEN <statement(s)1> ELSE
<statement(s)2>
```

Example 2.9

The following program illustrates the above points; study the structure then use it.

```
 10 REM IF...THEN...ELSE + FUNCTIONS
 20 MODE 7
 30 PRINT
 40 PRINT "To end run enter E or give a negative"'"angle"
 45 REM NOTE THE USE OF ' TO GIVE A NEW LINE
 50 PRINT
 60 REPEAT
 70 INPUT "Choose function sine, cosine or tangent by entering
    key letter only. ",A$
 80 INPUT "Give angle in degrees ",ANGLE
 90 ANGLE=RAD(ANGLE)
100 IF A$="S" THEN PRINT "RESULT ",SIN(ANGLE) ELSE IF A$="C"
    THEN PRINT"RESULT "COS(ANGLE) ELSE IF A$="T" THEN
    PRINT"RESULT ",TAN(ANGLE) ELSE IF A$="E" THEN END ELSE
    PRINT"Key letter not recognized use S, C or T"
110 PRINT
120 UNTIL ANGLE<0
130 END
```

The use of the character ' to give a new line is fully explained in Section 11.4.1. Extensive use can be made of IF...THEN...ELSE but like many good things it can be over indulged and Example 2.10 below, while valid, does not have adequate clarity – ON...GOTO would be more appropriate. Note that the THEN can be omitted but it is not recommended unless space is at a

premium. Also, in order to enter the line into the computer the abbreviation P. may have to be used.

Example 2.10

```
 5 REPEAT
10 INPUT "Choose an integer < 7 ",N
20 IF N/2<>INT(N/2) THEN P."Odd number":IF N=1 THEN P."One"
   ELSE IF N=3 THEN P."Three" ELSE IF N=5 THEN P."Five":ELSE
   P."Even number":IF N=2 THEN P."Two" ELSE IF N=4 THEN
   P."Four" ELSE IF N=6 THEN P."Six" ELSE P."Number out of
   range."
25 UNTIL N<0
```

Example 3.6 shows a version of the program using ON...GOTO/GOSUB. The use of the ON...GOTO statement is dealt with in Chapter 3.

2.2.3 The absent structured facility WHILE

An alternative construction to REPEAT...UNTIL in some languages is WHILE...ENDWHILE. Unfortunately BBC BASIC does not have this structure, which would have allowed the user to have a section of program that might not be executed at all but which on another occasion could be iterated many times. WHILE provides a facility that is not available with FOR...NEXT and REPEAT...UNTIL loops, which are always executed at least once. In a BASIC with a WHILE structure it is possible to write programs that do not need GOTO statements.

The construction of the WHILE structure is

```
100 WHILE <condition>
... \
... } Program statements of loop
... /
200 ENDWHILE
210 REM NEXT STATEMENT
```

The easiest way to achieve the same effect is

```
100 IF NOT <condition> THEN GOTO 210
110 REPEAT
... \
... } Program statements of loop
... /
200 UNTIL NOT <condition>
210 REM NEXT STATEMENT
```

2.2.4 The LISTO statements

LISTO, the list option statement, is a non-executable BASIC instruction which is particular to the BBC microcomputer. It affects the manner in which

the listing of a program is displayed on the video monitor or on a printer. It causes the computer to insert spaces in three situations either individually or collectively, as follows

1. After a line number.
2. To indent FOR...NEXT loops.
3. To indent REPEAT...UNTIL loops.

The mode of listing selected is indicated by a single number in the range 0 to 7 after the statement issued in immediate mode

```
LISTO 7 <RETURN>
```

(A syntax error occurs if it is used in a program.)

0 implies no inserted spaces
1 implies a space after the line number
2 implies spaces to indent FOR...NEXT loops
4 implies spaces to indent REPEAT...UNTIL loops

If several options are required they can be chosen at one time by adding the key numbers; hence 7 gives all options and zero none.

It should be noted that these are additional spaces to any that are stored as part of the text of the program. It is recommended that LISTO 7 is used to obtain listings which show clearly the structure of a program. If editing is done with LISTO 7 in force then additional spaces become part of the edited program. It is therefore advisable to return to LISTO 0 and relist the program before any editing is done. These LISTO options can conveniently be stored in the function keys by the immediate mode statements

```
*KEY0 LISTO 0 ||M ||0 <RETURN>  and
*KEY1 LISTO 7 ||M ||N LIST ||M
```

which enable the LISTO option to be changed by a single key stroke. The inclusion of <CTRL-N> and LIST in function key f1 is convenient since this sequence of commands are often used during the development of a program. The LISTO 0 option is in force after a <BREAK> sequence.

Exercise 2.5

Define the red function keys as suggested above and use them and the other options to LIST some of the example programs.

2.3 ARRAYS

2.3.1 Introduction

It is possible to represent a whole set of values by a single subscripted variable called an *array*. They are written in the form ARRAY(J) where the name of

the set of variables is ARRAY and J is the *subscript* that identifies a particular value of the set. Arrays are particularly useful in conjunction with FOR... NEXT loops, since every element of the array can be accessed in turn by treating the control variable of the loop as the array subscript.

The array ARRAY(J) must be declared once at the beginning of the program, before it is used, by a DIMension statement of the form

```
10 DIM ARRAY(10)
```

This sets aside eleven variables that can be identified by ARRAY(J) where J is an integer from 0 to 10 (the array subscripts always start from zero).

Example 2.11

The program below, which sums the first ten primes, has been written to illustrate the use of an array in a FOR...NEXT loop, where its use is particularly appropriate. Compare this program with that in Example 2.6.

```
 10 DIM PRIME(10)
 20 SUM=0: REM INITIALIZE SUM
 30 FOR J=1 TO 10: REM ITERATION TO READ DATA
 40 READ PRIME(J): REM OBTAIN PRIME FROM DATA
 50 NEXT J: REM LOOP BACK
 60 FOR J=1 TO 10: REM ITERATION TO FORM SUM
 70 SUM=SUM+PRIME(J): REM ADD TO PREVIOUS SUMMATION
 80 NEXT J: REM LOOP BACK
 90 PRINT "Sum of first 10 primes = ";SUM: REM PRESENT RESULT
100 END
110 DATA 1,2,3,5,7,11,13,17,19,23
```

2.3.2 Multiple-subscript arrays

An array may have more than one subscript; for instance a set of x- and y-coordinates to be plotted on a graph can be stored in the array defined by

```
10 DIM XY(2,20)
```

where XY(1,J) represents the x-coordinates and XY(2,J) represents the y-coordinates. In fact, the array has a set of variables XY(0,J) available since the BBC microcomputer always allocates storage locations for an array from 0 to the values specified in the dimension statement. There is no limit to the number of subscripts an array may have; nor is it necessary to use all the elements, but remember that it can be extremely wasteful of memory to create multiple-subscript arrays which are too large.

For example the program

```
20 DIM A(10,10,10)
30 MODE 1
```

cannot be run successfully on a computer with a disc filing system fitted, with the normal PAGE setting! The error message

```
Bad MODE at line 30
```

is generated when the high resolution four colour graphics mode, which requires 20K of memory, is selected. If a 10×10×10 array is needed remember that the elements with subscript zero always exist, and line 20 should then be

```
20 DIM A(9,9,9)
```

which will enable the two line program to run successfully. In fact another 867 bytes are available for the remainder of the program.

The multiplication table program in Example 2.12 illustrates the use of a two-subscript array to store a table of values. In this case the table stored is the multiplication table from 1×1 to 12×12; a trivial example, which it would be more appropriate to calculate as required, but it shows how a table of values can be stored and accessed. This would be valuable if the data took a long time to calculate, (for example tables of values of sin and cos for a rotation program), or if the data had been obtained from measurements or observations and was needed within the program.

Lines 30 to 100 set up the table and print it out as it is formed. Lines 110 to 130 enable the table to be read, and line 140 permits the execution of the program to be terminated if a zero is entered.

Note that a correct answer is given for a product involving zero as array variables are all set to zero when the DIM statement is executed. Thus array variables can appear on the right hand side of an assignment statement before they have had a value assigned to them, in contrast to simple variables.

Example 2.12

```
 10 REM MULTIPLICATION TABLES
 20 DIM M(12,12)
 30 FOR J=1 TO 12
 40 FOR K=1 TO 12
 50 M(J,K)=J*K
 60 PRINT M(J,K);
 70 NEXT K
 80 PRINT
 90 PRINT
100 NEXT J
110 REPEAT
120 INPUT "Type 2 Numbers Under 13 ",A,B
130 PRINT ;A;" * ";B;" = ";M(A,B): PRINT
140 UNTIL A=0 OR B=0
```

Exercise 2.6

Write a program to read in and then calculate the product of a set of numbers. (Hint: the initial value of the product must be 1, in contrast to the starting value of 0 for a sum.)

2.3.3 String arrays

String variables, like real variables, may have subscripts and be expressed in arrays. String arrays are denoted by the addition of the dollar sign, $, to the array variable name. Thus DIM NAME$(7) permits the setting up of an array of eight string variables. As string arrays will have to store strings of variable lengths (up to a maximum of 256 characters), no space is allocated for the string at the time the array is declared. The DIM statement merely allocates space for address pointers to the beginning and end of each string (four bytes for each element of the array). The consequence of the need for a large amount of additional space for the strings themselves, whilst a program is running, is that the computer may run out of room during the execution of the program. This causes the error message

```
No room at line XX
```

to be generated.

For example, the program

```
10 DIM A$(6326)
20 INPUT A$(1)
```

will work if the string given is two characters or less, but will give the error message

```
No room at line 20
```

if the string has three or more characters. After the program has been run it will be found that the command LIST gives the error message 'No room'. This is because memory is still allocated to the program. Typing CLEAR or OLD followed by LIST will give a listing of the program. (The size of the array chosen here is for a Model B microcomputer with DFS and default PAGE setting of &1900. PRINT ~PAGE will confirm the setting of PAGE.)

Clearly, if space is likely to be a problem (in programs using graphics modes, for example), it is advisable to declare only the minimum possible amount of string array space and to use it sparingly. In programs involving a large number of user-friendly prompts it may be advisable to keep the text in DATA statements and only read it when necessary. This means that text is not stored twice as it is when it is assigned to a string variable by an assignment statement. If this ploy does not give enough space then user-friendly material could be stored in a ASCII file for display at appropriate times.

Another memory problem occurs if short strings are later replaced by longer strings which cannot be stored in the same positions in memory. New space is allocated to them further on in the memory but the old space remains unused and is therefore wasted. If this is likely to occur it is better to allocate long strings – for example a string of 20 spaces, STRING$(20," ") – to each element of the array at the beginning of the program, even though this uses a lot of memory. At least, if this does run out of memory the error message will occur right at the beginning of the program and not half way through after a lot of data has been entered.

Chapter 3

STRUCTURED PROGRAMMING FACILITIES

3.1 INTRODUCTION

3.1.1 Structure charts (top-down design)

The BBC microcomputer has a wide range of facilities that offer the programmer the opportunity to write clear and easily followed programs. To make the best use of this potential it is worth spending some time in careful analysis of the problem that is to be programmed in order to organize the program in the most appropriate way. This analysis should be done before any of the BASIC code is written. It will be time well spent and if the decisions reached are recorded they will be useful in making modifications to the program at a later date. There are two main techniques for recording the decisions about the form of the program: the *flow chart*, and the *structure chart* produced in *top-down design*.

It is now accepted that top-down design is a method of structured programming that can lead to good programs. The name comes from the idea of considering a problem as a whole, then breaking it down into a small number of main sections, and then progressively breaking each section down into smaller sub-sections until each sub-section can readily be coded in the programming language to be used. The sections specified are recorded in a structure chart (not to be confused with a flow chart), which should show the overall structure of a program in diagrammatic form. The structure chart should be easily understood and so should not contain complex flow lines that belong in flow charts.

There are three main logic structures which may be represented in a structure chart: sequence, iteration and selection. It is appropriate to consider each of these in turn.

Sequence

The first stage in the analysis of a problem that is to be programmed is to break it down into a small number of identifiable sections. At this stage of analysis the natural sequence of different processes provides an obvious

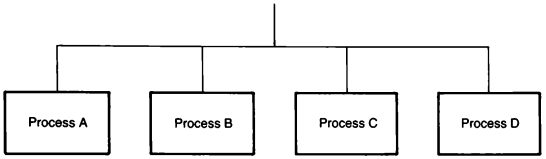


Figure 3.1 The use of sequence boxes in a structure chart.

division of the problem. Each of these is identified by a rectangular box called a sequence box. Figure 3.1 illustrates the use of sequence boxes.

The boxes should be read from left to right; the line links them to show they form the subdivided parts of a whole unit. Thus process A is followed by process B which in turn is followed by process C and last comes process D.

Iteration

An iteration is a process which involves repetition (possibly of a number of sequence events). A process which is repeated is represented in a structure chart by a rectangular box with an asterisk in the top right hand corner (see Figure 3.2). The number of times that the process is repeated is not necessarily indicated.

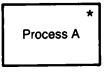


Figure 3.2 Sequence box for a repeated process.

Selection

If only one process is to be chosen from two or more possible processes a selection construction is used. The processes may be regarded as a number of

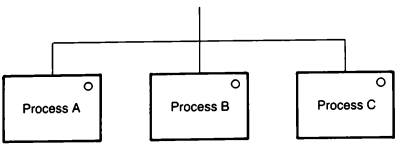


Figure 3.3 Representation of a selection group.

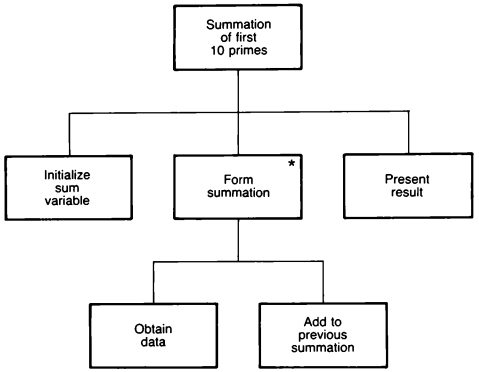


Figure 3.4 Structure chart for summation of primes.

options. The selection construction is represented by a series of rectangular boxes each with a small circle in the top right hand corner. The group from amongst which the choice is made is connected by a line but in this case only one process is executed. Figure 3.3 illustrates such a selection group.

A structure chart is built up using the three key structures. It starts with a single box indicating the whole program. Examine the program to sum the first 10 prime numbers (Example 2.6) and its structure chart in Figure 3.4.

The structure chart in Figure 3.5 is for Example 2.11 in which an array is used to solve the same program requirement. Note how much easier it is to

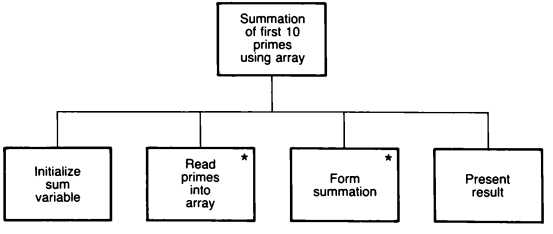


Figure 3.5 Structure chart for summation of primes using an array.

follow and consequently the case for the use structure charts can be seen. It is worth pointing out that these programs are really too simple to need a structure chart.

3.1.2 Good programming style

It is important that even the relative newcomer to programming seeks to adopt a systematic, coherent and professional approach to the design and implementation of computer programs.

There are general rules which can be applied by the BASIC programmer which will reduce the chances of error and confusion. Their importance will become more apparent as the sophistication of the problems tackled increases.

1. First analyse the problem systematically (before you start typing!) Decide on the best strategy to solve the problem.
2. Break the problem down into a series of simpler parts (modules). Each module might well correspond to one sequence box of the structure chart. Wherever possible, particularly in an extensive program, write each substantial module as a procedure. One of the great advantages of BBC BASIC is the provision of proper procedures and functions, in addition to the standard subroutines. Each procedure, function and subroutine should be separately tested and verified.
3. As an aid to clarity, use variable names which will remind anyone reading the program of the actual quantities to which they refer. Lower case can be used for variable, procedure and function names and this helps in distinguishing them from keywords. However, it makes typing more difficult, with the need to keep operating the CAPS LOCK key. (In this book, a convention has been used of lower case for procedure and function names, but upper case for variable names).
4. For the same reason, use REM statements to explain the function of any potentially unclear lines or parts of the program.
5. Try to avoid using GOTO instructions unnecessarily. This is one of the more unfortunate features of BASIC where some thought will often reveal a better way to write the program without them. Extensive use of GOTO leads to programs which are very difficult to check and understand. Consider it as a challenge to keep GOTOs out of your programs.
6. In the case of extensive programs, the analysis of stages (1) and (2) should be separately documented.

3.2 FUNCTIONS

BASIC has many built-in functions, such as SIN(X), COS(X), RND(X), INT(X) and LOG(X). In addition, it is possible for the user to define his own functions, each consisting of a separate section of program which returns a single value to the calling statement in the main program. Functions created

by the user are used in exactly the same way as the built-in functions, but they are preceded (without any intervening spaces) by the letters FN. For example, function 'round2' would be called by

```
10 MONEY=FN_round2(X)
```

The underline character is used here, and throughout this book, to give a visual impression of a space between FN and its name since the space character cannot be used. (The same convention is followed for procedures.)

Functions may be defined outside the main body of the program and it is conventional to collect all the functions (and procedures) at the end of a program, after the final END statement. In any case they must not be in a part of the program where the lines comprising the function would be executed other than due to a function call.

A function is defined by a section of program that starts with a line such as

```
100 DEF FN_round2(R)
```

and ends with a line such as

```
140 =X
```

In BBC BASIC a function may contain several statements and can be as long as necessary. The variable R used in the definition of a function is called the *formal parameter*. The value or expression used when the function is used is called the *actual parameter* and has no relationship to the formal parameter. The value of a variable used as an actual parameter is unchanged after use of the function even if the value of the formal parameter is reassigned within the function. Functions may have more than one parameter, separated by commas.

Example 3.1

This program uses a function FN_round2 which rounds a given number to two decimal places. This might be a useful function in a program involving monetary calculations or for rounding off average or scaled exam marks. The function is called at line 50 and the rounding to two decimal places is done by multiplying by 100 and then taking the integer part of the result with the INT(X) function. The integer result is then divided by 100 to get the required two decimal place form of the number.

```
10 REM TO ILLUSTRATE THE USE OF DEF FN
20 MODE 7
30 REPEAT
40 INPUT "Give a number to more than 2 decimal places ",R
50 MONEY=FN_round2(R)
60 PRINT "To the nearest penny £";MONEY
```

```
 70 PRINT "Do you wish to do a new rounding?"'"Answer Y or N."
 80 UNTIL NOT (GET$="Y")
 90 END
100 DEF FN_round2(R)
110 REM ROUND A GIVEN NUMBER TO 2 DECIMAL PLACES
120 X=INT(R*100+0.5)/100
130 =X
```

The function definition is spread over several lines for clarity, but if necessary this particular definition could be in one line as

```
100 DEF FN_round2(R)=INT(R*100+0.5)/100
```

(There is another way of achieving this presentation with @%, which is the print format control parameter – see Chapter 11.)

In BBC BASIC a function can have several parameters so that the distinction between procedures and functions is blurred. However, only the function can return a value without the need for a common variable. (To return more than a single variable you would clearly have to use a procedure.)

Example 3.2

The following program uses a function closely related to FN_round2, together with two of our other structures, IF...THEN...ELSE and REPEAT...UNTIL.

```
 10 REM TO ILLUSTRATE VARIOUS PROGRAMMING STRUCTURES
 20 MODE 7
 30 REPEAT
 40 FLAG=0: REM USED TO INDICATE ROUNDING CAN BE DONE (FLAG=0)
 50 INPUT "Give a number to more than 2 decimal places ",R
 60 INPUT "How many decimal places ",NDP
 70 IF (NDP=2) THEN MONEY=FN_round(R,2) ELSE IF NDP=3 THEN
    MONEY=FN_round(R,3) ELSE FLAG=1
 80 IF FLAG=0 THEN PRINT "To ";NDP;" decimal places "; MONEY
    ELSE IF NDP=INT(NDP) THEN PRINT "Out of range of available
    functions." ELSE PRINT "The number of decimal places
    required"'"must be given as a positive integer."
 90 PRINT "Do you wish to do a new rounding?"'"Answer Y or N."
100 UNTIL NOT(GET$="Y")
110 END
120 DEF FN_round(R,N)
130 REM ROUND A GIVEN NUMBER TO N DECIMAL PLACES
140 X=INT(R*10^N+0.5)/10^N
150 =X
```

In this program a variable FLAG is used to indicate whether rounding can be done. A value of zero for FLAG means that the function can be used, and at

line 70 FLAG is set to 1 if FN_round can not be used. Try running the program with a range of values for R and NDP including non-integer values for NDP. The program should cope with this unreasonable data. Note that FN_round allows rounding to any number of decimal places specified by NDP. The IF...THEN statement at line 70 limits the use FN_round in this program. This restriction has been introduced deliberately to illustrate the use of a function within an IF...THEN statement.

Exercise 3.1

Write a program to define functions for the hyperbolic sine and cosine and test, for a range of values of x, the quantity

```
FN_hcs(X)*FN_hcs(X) - FN_hsn(X)*FN_hsn(X)
```

where functions FN_hcs and FN_hsn are defined by

```
FN_hcs(X) = (EXP(X)+EXP(-X))/2
FN_hsn(X) = (EXP(X)-EXP(-X))/2
```

Exercise 3.2

Given that 0°C corresponds to 32°F and that 100°C corresponds to 212°F, devise a function to convert centigrade to Fahrenheit and another one to convert Fahrenheit into centigrade.

3.3 LOCAL PARAMETERS

It is possible to define variables used in a function to be LOCAL to that function so that variables in the main part of the program are not corrupted by use of the same variable name in a function. Such variables must be declared as local by a line such as

```
110 LOCAL IPART,DPART
```

Any variable not declared as LOCAL is 'common' or 'global'; that is, it takes the value assigned in the function when next used in the main program; or when used on the right hand side of an assignment in the function it retains any previously set value.

The use of functions is not restricted solely to where a value is to be returned to the calling program. The value returned can be ignored or given to a variable such as DUMMY which is not used elsewhere in the program.

Example 3.3

This program uses two functions FN_delay and FN_response. FN_delay produces a random delay up to a maximum value by using the standard RND function and the psuedo-variable TIME. The function FN_response is used

to measure the time between the random display of a random character and the user response by pressing the selected key. The FN_response also uses the RND function. Functions can be nested 26 deep. Examine the program and note that the variable START is used in both functions but is declared to be LOCAL. Try printing START in immediate mode after the program has been run.

```
 10 REM FN_response
 15 MODE 7
 20 REPEAT
 30 CLS
 40 PRINT "Press the letter given below as quickly as possible
    after it is displayed"
 50 REACTION-TIME=FN_response(A$)
 60 PRINT: PRINT "You took ";REACTION-TIME;" seconds to press
    ";A$
 70 PRINT "For another trial press Y "
 80 A$=GET$
 90 UNTIL A$<>"Y"
100 END
110 DEF FN_delay(CENTISEC)
120 LOCAL START
130 START=TIME
140 REPEAT
150 UNTIL (CENTISEC+START)<TIME
160 =CENTISEC
170 DEF FN_response(A$)
180 LOCAL D,START,B$,X,Y
190 DUMMY=FN_delay(RND(150))
200 A$=CHR$(64+RND(26))
210 START=TIME
220 X=RND(39)
230 Y=3+RND(15)
240 REPEAT
250 PRINT TAB(X,Y);A$
260 B$=GET$
270 UNTIL B$=A$
280 =(TIME-START)/100
```

Try the effect of adding the line

```
215 FN_delay(100)
```

This has the effect of using the LOCAL variable START in each function at the same time so that it has different values in each function. This is permitted since START is LOCAL in both functions. This alteration will mean that the minimum response time will now be one second.

3.4 PROCEDURES (DEF PROC)

Procedures are similar in some respects to functions. They do not, however, appear at the right hand side of expressions and do not return values to the calling program as functions can do. There are two important restrictions on the use of procedures in BBC BASIC. Firstly, only common variables can be used to transfer values back to the main program. Secondly, whole arrays can not be used as parameters, although array elements may be used as actual parameters when a procedure is called. Procedures are defined by lines such as

```
150 END
200 DEF PROC_nice_one(FRED,JIM,ETC)
    ...
    ...
    ...
240 ENDPROC
```

They are used within a program by a 'call' of the form

```
70 PROC_nice_one(ALICE,JOAN,ANON)
```

Procedure definitions should be outside the main program and are usually placed after END along with the function definitions. As with functions, variables used in procedures are held to be the common variables in the main program unless they are specifically declared as LOCAL. The procedure may have any number of formal parameters or none at all. The actual parameters are unchanged after the procedure has been called.

Exercise 3.3

The program in Example 3.3 uses a function, FN_delay, which does not return a value to the calling program. It is thus appropriate to rewrite this program to use a procedure to give the required delay. After you have made and tested this modification alter the procedure to give a one second delay without the need for a formal parameter.

Example 3.4

```
 10 REM DEF PROC DEMONSTRATION PROGRAM
 20 CLS: PRINT
 30 READ FRED,JIM,ETC
 40 INPUT "Three ages in years < 70 ",ALICE,JOAN,ANON
 50 SUM=FRED+JIM+ETC
 60 PRINT "Sum of men's ages = ";SUM
 70 PROC_nice_one(ALICE,JOAN,ANON)
 80 PRINT "Return to main program "
 90 PRINT "FRED = ";FRED
100 PRINT "JIM  = ";JIM
```

```
110 PRINT "ETC  = ";ETC
120 PRINT "Sum of women's ages = ";SUM
130 PROC_nice_one(RND(70),RND(70),RND(70))
140 PRINT "Sum of 3 random ages = ";SUM
150 END
200 DEF PROC_nice_one(FRED,JIM,ETC)
210 PRINT "Procedure NICE_ONE entered"
220 PRINT "INITIAL VALUES ";FRED,JIM,ETC
230 SUM=FRED+JIM+ETC
240 ENDPROC
250 DATA 27,23,30
```

In this example FRED, JIM and ETC are the formal parameters of the procedure, and all formal parameters of a procedure are LOCAL to it even if FRED, JIM and ETC are also in the main program. One rather serious restriction, as already mentioned, is that it is not possible to return values through the formal parameters as is the case in languages such as Pascal and FORTRAN. Values to be used subsequently in the main program must be passed through common variables such as SUM. These features of procedures are illustrated by this trivial example. At line 30 FRED, JIM and ETC are given values in the READ statement. Lines 90 to 110 demonstrate that these values are retained even when the procedure is called, as the formal parameters are automatically LOCAL. The variable SUM is calculated at line 50 and printed at line 60. The variable SUM is common to the main program and the procedure so that at line 120 when it is printed again it will have a value determined by the procedure. The formal arguments of the procedure may be variables, as at line 70, or arithmetic or other valid statements as at line 130 where the RND function is used.

Example 3.5

The procedure in this example is used to solve the quadratic equation $ax^2+bx+c=0$. It shows a more useful application of a procedure within an IF...THEN...ELSE, where GOTOs would otherwise have been needed. It would of course be possible to include the IF...THEN statement within the procedure, but the point here is to illustrate the use of a procedure call in an IF...THEN statement.

```
10 REM SOLUTION OF QUADRATIC EQUATIONS
20 MODE 7
30 PRINT
40 PRINT "Solution of quadratic of the form:"'"A*X*X + B*X +
   C = 0"
50 REPEAT
60 PRINT
70 INPUT "Coefficients of quadratic A,B and C separated by
   ','."',A,B,C
80 REM USE OF CONDITIONAL TEST TO PREVENT PROGRAM FAILURE ON
   EVALUATION OF NEGATIVE ROOT
```

```
 90 IF B*B<4*A*C THEN PRINT "No simple roots": PRINT ELSE
    PROC_quad(A,B,C)
100 PRINT "Do you wish to give new data? ";
110 A$=GET$
120 PRINT
130 UNTIL A$<>"Y"
140 END
150 DEF PROC_quad(AQ,BQ,CQ)
160 ROOT=SQR(BQ*BQ-4*AQ*CQ)
170 PRINT "Roots are ";(-BQ+ROOT)/(2*AQ);" AND ";
    (-BQ-ROOT)/(2*AQ)
180 ENDPROC
```

Exercise 3.4

Write a procedure to examine a list of 10 numbers stored in an array, find the largest and the smallest, and return them in common variables LOW and HIGH. Test the procedure in a simple program.

Exercise 3.5

Write a procedure to evaluate the first N prime numbers and store them in an array PRIME(J) of dimension N. Incorporate the procedure in a program to calculate the sum of the first N prime numbers, where N is an integer supplied by the user. (Hint: write and test the procedure for the prime numbers then look at Chapter 2 for the summation program using an array.)

3.5 OTHER PROGRAM STRUCTURES

3.5.1 Subroutines (GOSUB)

It is usually preferable to use procedures rather than GOSUB in BBC BASIC, since they are available. However, the BBC BASIC supports subroutines so that programs taken from listings for other computers can still be run.

The GOSUB facility has some attractions in that the routine can be entered at any point. However, subroutines do not have the possibility of LOCAL variables or formal parameters.

3.5.2 Computed GOTO and GOSUB (ON...GOTO and ON...GOSUB)

The usual numerical sequence in which the BASIC program statements are executed may be modified not only by REPEAT...UNTIL and GOTO, but by the statement

```
10 ON <value> GOTO XX,YY,ZZ
```

where XX, YY and ZZ are line numbers. If the <value> is 1 then the effect is to GOTO XX. If the <value> is 2 then the program goes to YY. Finally if the

<value> is 3 then the effect is to GOTO ZZ. If the <value> is less than 1 or exceeds 3 then an error occurs.

Strictly the values for 1, 2 and 3 can lie in the range

```
0.9999999999  to 1.99999999975
1.99999999976 to 2.99999999953
2.99999999954 to 3.99999999952
```

respectively on input of data. It is left as an exercise for the reader to determine the range for higher switches.

Example 3.6

The program structure chart in Figure 3.6 illustrates the program below, which demonstrates a simple ON...GOTO statement. The user is asked to supply a number at line 20. This should be 1, 2 or 3. The program then prints ONE, TWO or THREE as appropriate by use of the ON...GOTO statement at line 30. Examine the program and run it.

```
 10 REPEAT
 20 INPUT "Choose a value for I ",I
 30 ON I GOTO 40,60,80
 40 PRINT "Value of I = ONE"
 50 GOTO 90
 60 PRINT "Value of I = TWO"
 70 GOTO 90
 80 PRINT "Value of I = THREE"
 90 UNTIL FALSE
100 END
```

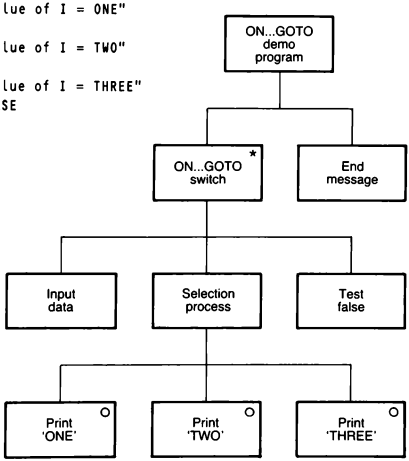


Figure 3.6 Structure chart for the ON...GOTO demonstration program.

Try values for I of 1, 2, 3, non-integer values less than 4 and then a value greater than 4. In the latter case you will get the error message

```
ON range at line 30
```

This can be overcome by modifying line 30 to be

```
30 ON I GOTO 40,60,80 ELSE 90
```

Note: the use of ELSE with ON...GOTO upsets the return from functions and procedures in the first version of BASIC (BASIC I), and so must not be used with a function or procedure. An alternative which overcomes the problem on BASIC I is to use the ON ERROR GOTO statement by adding the line

```
25 ON ERROR GOTO 20
```

and leaving line 30 as

```
30 ON I GOTO 40,60,80
```

Remember to turn the error trapping off with ON ERROR OFF when the range of this REPEAT sequence is ended. Note that ON ERROR GOTO 90 will not work since you cannot return into a FOR...NEXT or REPEAT...UNTIL loop, (in fact this generates error 43, No REPEAT) but note that it is permissible to add the line

```
5 ON ERROR GOTO 10
```

Exercise 3.6

Write a program that asks the user the time (on a 24 hour clock) and then responds with the appropriate message from the list GOOD MORNING, GOOD DAY, GOOD AFTERNOON, GOOD EVENING or GOOD NIGHT (in sequence through the 24 hours).

Exercise 3.7

Write a program to calculate the time in hours, minutes and seconds from the psuedo-variable TIME.

Exercise 3.8

Write a program to produce a calendar for any year since the year 1800 given that the first of January in that year was on a Wednesday.

3.6 RECURSIVE USE OF PROCEDURES AND FUNCTIONS

The BBC microcomputer permits the recursive calling of procedures and functions. This is valuable for the evaluation of a mathematical recursion formula in which successive terms of a series can be expressed in terms of

previous terms, and for other problems of this type. For example, the set of natural numbers, the integers, may be defined recursively in the following way.

1. 1 is a natural number.
2. The successor of a natural number is a natural number, where the successor of a number is obtained by adding one to that number.

The power of the recursive definition is its ability to define a set consisting of an infinite number of elements using only two statements. In a similar way a recursive procedure or function can specify an infinite number of computations using a few program statements and without looping. Using recursion in solving a problem often leads to a new problem which is similar to the original but smaller in scope.

Example 3.7

A simple example of a recursive solution is the computation of the factorial of a non-negative integer N. The non-recursive solution is given in the program below. This program uses a FOR...NEXT loop to evaluate N!

```
10 REM FACTORIAL EVALUATION
20 MODE 7
30 PRINT "Evaluation of the Factorial for integers Give a
   positive number less than 33 and a NEGATIVE number to
   finish."
40 PRINT: INPUT "Remember to give a POSITIVE No. < 33 ";X
50 IF X>0 THEN FAC=1: FOR I=1 TO X:FAC=I*FAC: NEXT ELSE END
60 PRINT "FACTORIAL ";X;" IS ";FAC
70 GOTO 40
80 END
```

Now consider the recursive solution. The stopping or end state is considered first.

1. If N is zero, the factorial is 1.

Next, the recursion step is considered.

2. For N>1, the solution is N times the factorial of (N−1).

Thus the recursion step has reduced the problem to that of finding the factorial of the next smaller number and multiplying that result by N. Obviously, N−1 is closer to the stopping state of zero than N is. repeated applications of the recursion step will eventually lead to the stopping state.

Example 3.8

The implementation of this in BBC BASIC is shown in the following program.

```
10 REM FACTORIAL EVALUATION
20 MODE 7
```

```
30 PRINT "Evaluation of the Factorial for integers"'"Give a
   positive number less than 33 and a NEGATIVE number to
   finish."
40 PRINT: INPUT "Remember to give a POSITIVE No. < 33 ";X
50 IF X>0 THEN FAC=FN_fac(X) ELSE END
60 PRINT "FACTORIAL ";X;" IS ";FAC
70 GOTO 40
80 END
90 DEF FN_fac(I)
100 LOCAL F
110 IF I<>0 THEN F=I*FN_fac(I-1) ELSE F=1
120 =F
```

The function FN_fac returns a value of 1 if its argument is zero. If the argument is non-zero then the value returned is determined by the expression N*FN_fac(N−1). This calling of FN_fac from within itself is what is termed recursion.

In evaluating factorial N each new call of FN_fac increases the level of recursion. The BASIC interpreter saves the values of the function's parameter N and of any local variables just before each new call. At the time of the call, the N−1 is calculated and becomes the parameter N for the next call of FN_fac. Clearly the value of N will be zero for the last call of the function. As the interpreter returns through each level of recursion, it evaluates the value returned by multiplying the parameter saved at this level, by the result just returned from the level below. Each return reduces the level of recursion by one. When the level of recursion is back to zero the value returned represents the value of the initial function call.

Figure 3.7 shows the case of 6! (factorial 6). There are 7 levels of recursion (0 to 6) and 6 recursive calls of FN_fac plus the initial call. The arrows indicate the flow of the calculation.

Level of recursion	*What is required*	*What is actually calculated*
0	6!	720
	↓	↑
1	6×5!	6*120
	↓	↑
2	5×4!	5*24
	↓	↑
3	4×3!	4*6
	↓	↑
4	3×2!	3*2
	↓	↑
5	2×1!	2*1
	↓	↑
6	1×0! →	1

Figure 3.7 Action of a recursive function.

Example 3.9

Consider the problem of obtaining the digits of a number printed in reverse order. The recursive solution is as follows

1. Output the last digit of the number.
2. If further digits remain then 'reverse' remaining digits, otherwise print the remaining digit.

The recursive step reduces the number of digits that must be reversed by one each time until just one remains which needs no reversal. The procedure is used to generate the PRINT statements which give the reversed number. This example has been written as a recursive procedure since the number of levels of recursion is not of great interest and thus there is no need to return a result to the calling program.

```
 10 REM Number reversal example
 20 REPEAT
 30 PRINT: PRINT "To finish type a negative number"
 40 INPUT "Number to be reversed ",N
 50 PRINT ;N;" reversed is ";
 60 PROC_reverse(N)
 70 PRINT
 80 UNTIL N<=0
 90 END
100 DEF PROC_reverse(N)
110 IF N DIV 10 = 0 THEN PRINT ;N;:F=1 ELSE PRINT ; N MOD 10;:
    PROC_reverse(N DIV 10)
120 ENDPROC
```

Example 3.10

Suppose that it is desired to obtain a number of permutations of the integers 1 to 4. The function RND(N) can be used to select the order but each time a number is chosen there is one less to choose from next time, until there is no choice. The program below shows a non-recursive solution.

```
 10 REM for Linda Leech
 20 PRINT"RANDOM PERMUTATIONS OF NUMBERS 1 TO 4"
 30 DIM N(4),M(4)
 40 FOR L=1 TO 20
 50 FOR I=1 TO 4:N(I)=I: NEXT
 60 FOR I=4 TO 2 STEP -1
 70 J=RND(I)
 80 M(I)=N(J)
 90 N(J)=N(I)
100 NEXT I
110 M(1)=N(1)
120 PRINT M(1);"  ";M(2);"  ";M(3);"  ";M(4)
130 NEXT L
```

Two arrays M and N are used in this program. The first FOR...NEXT loop allows 20 permutations to be selected. The FOR...NEXT loop at line 30 fills the N(I) array with the numbers 1 to 4 in initial order. The permutation loop begins at line 40 and at line 50 a random number between the count variable I and 1 is chosen by the RND(N) function. This number is used to choose the first number of the permutation which is stored in the Ith element of the M array. At line 70 the N array is changed to take account of the number that has been selected in the first permutation, and then I is decremented and the next number selected. This process continues until just one number remains. The last number is transferred to the M array at line 90 and at line 100 the array is printed for the permutation. This process in lines 30 to 100 is repeated 20 times.

Example 3.11

The recursive solution is shown in the solution below. In this example of recursion a procedure is used rather than a function. The arrays M and N are common to the main program and the procedure.

The recursive solution is as follows

1. Find the next integer for the permutation.
2. If only one integer remains then this is the last integer for the permutation.

This procedure is called at line 40. The two arrays are used in the same way as before.

```
 10 REM RECURSIVE PERMUTATIONS
 20 PRINT"RANDOM PERMUTATIONS OF NUMBERS 1 TO 4"
 30 DIM N(4),M(4)
 40 FOR L=1 TO 20
 50 FOR I=1 TO 4:N(I)=I: NEXT
 60 PROC_comb(4)
 70 PRINT L;"    ";M(1);"    ";M(2);"    ";M(3);"    ";M(4)
 80 NEXT L
 90 END
100 DEF PROC_comb(I)
110 IF I<>1 THEN J=RND(I):M(I)=N(J):N(J)=N(I): PROC_comb(I-1)
    ELSE M(I)=N(1)
120 ENDPROC
```

Exercise 3.9

Write a program using a recursive procedure to select combinations of 3 integers from 6.

Chapter 4

STRINGS

4.1 STRING VARIABLES

A string variable enables the programmer to handle strings of alphabetic, numeric and other characters in a way that is very similar to the use of numeric variables for numbers. A string variable is denoted by a final $ sign in the variable name. String variables such as A$, NAME$, Text$ etc. may have character strings assigned to them in the same way that numbers are assigned to ordinary variables, except that in assignment statements the character string must be enclosed in quotation marks. Like numeric variables, string variables must be assigned before they are used. The PRINT and INPUT commands can be used for string variables.

Simple string variables are very useful in making programs 'user-friendly'. For example, the following exercise uses a string variable to store the user's name.

Example 4.1

```
 10 CLS
 20 PRINT "Please give your name"
 30 INPUT NAME$
 40 PRINT "If you want help, ";NAME$;" type 1, otherwise type
    2"
 50 INPUT X
 60 IF X=1 THEN PROC_help
 70 REM
 80 REM FURTHER SECTIONS OF PROGRAM WOULD BE INSERTED HERE
 90 REM
100 PRINT "Please give your age, ";NAME$
110 INPUT AGE
    ...
    ...
    ...
    ...
```

```
 990 END
1000 DEF PROC_help
1010 PRINT "Help information"
1020 REM HELP INSTRUCTIONS WOULD BE
1030 REM INSERTED HERE
1070 PRINT "Press any key to continue"
1080 B$=GET$
1090 ENDPROC
```

4.2 PRINTING WITH STRINGS

4.2.1 INPUT

After the INPUT command, one or more strings of text between quotation marks can be printed as a prompt before any of the variables to be input. Each string of text and final quotation mark must be followed by a comma. Note that a string variable *cannot* be used instead of a string in this context, since it would be interpreted as a variable to be input. (In BASIC II *only*, semicolons may also be used as separators, both after the strings of text and between variables. The latter option makes BBC BASIC compatible with 'standard' BASIC, where a semicolon is used after a string of text, and commas between variables.)

Exercise 4.1

Modify Example 4.1 as follows

```
100 INPUT "Please type your age ",AGE
110 PRINT "Thank you ";NAME$
```

The data is entered on the same line as the prompt printed in the INPUT statement, which leads to greater clarity in programs.

The same effect can be achieved by typing a semi-colon at the end of the PRINT statement, so that the cursor doesn't move on to a new line.

Exercise 4.2

Modify Exercise 4.1 as follows

```
 95 C$=" please give your age "
100 PRINT NAME$+C$;
105 INPUT AGE
```

The question mark printed on the screen when an input is wanted can be unnecessary and irritating if you are using prompts. It can be suppressed by leaving out the comma after the string at the start of the INPUT statement. This can be done even with an 'empty' string, so that Exercise 4.2 could be modified to

```
105 INPUT ""AGE
```

4.2.2 Concatenation of strings

As demonstrated above, strings can of course be concatenated, or 'added' together in a PRINT statement. It is also possible to 'add' strings within a program and assign new values to a string by statements such as

```
X$=A$+B$+C$
```

Note, however, that no other arithmetic operations (not even subtraction) are possible with string variables.

Exercise 4.3

To demonstrate this, modify Exercise 4.1 again as follows

```
 95 C$=NAME$+" please give your age"
100 PRINT C$;
```

4.2.3 INPUT/READ statements for string variables

In assignment statements for strings, it is essential to use quotation marks, for example

```
A$="A word"
```

In an INPUT or READ statement the quotation marks are optional so long as it is not necessary to input a comma or semicolon. With the READ statement, if leading spaces, commas or semicolons are to be read in, it is essential for the whole string to be enclosed in quotation marks. On the other hand, quotation marks can be included in strings that start with some other character than a quotation mark (single inverted commas can be freely input in READ or INPUT responses). It is actually possible to input a quotation mark anywhere within a true string by typing a *pair* of quotation marks.

The same technique can be used with the INPUT statement, but there is also an easier alternative. An extra item can be included within the items after INPUT. This is the word LINE, which must appear at the start of the INPUT statement; its effect is to take the whole line of input that is given in response to the statement and put it into the next variable. This gives a simple means of including leading spaces, commas and quotation marks anywhere in a string. Obviously LINE is only useful in connection with string variables.

The following statement would be quite legitimate in BBC BASIC

```
INPUT LINE,"Length of strings",L,"First string",A$, "Second
  string",B$
```

Exercise 4.4

Verify the statements above by experimenting with different responses to the following program

```
10 CLS
20 INPUT LINE,X$,Y$,Z$
30 PRINT X$,Y$,Z$
```

Modify the program to use READ and DATA statements

```
20 READ X$,Y$,Z$
40 DATA "WHY, OH WHY",SAY "AH","""AH""": REM TEST DATA
```

4.3 COMPARISON OF STRING VARIABLES

It is possible to compare string variables for equality in an IF statement. The result will be TRUE only if there is complete equality – a character by character correspondence between two strings of equal length. It is also possible to test for inequality (<>) and greater than (>) or less than (<).

Example 4.2

The following section of program shows a test for the answer to a question.

```
10 CLS
20 Y$="YES"
30 INPUT "Do you want to continue? "YN$
40 IF YN$="NO" THEN PRINT"Program ended": END
50 IF YN$<>Y$ THEN PRINT "Please type YES or NO": GOTO 30
60 PRINT "Program continued"
70 REM FURTHER SECTIONS OF PROGRAM WOULD GO HERE
```

4.3.1 Alphanumeric ordering

String variables can be sorted into alphanumeric order by use of IF statements of the form

```
IF A$<=B$ THEN...
```

The IF test will be TRUE if a character by character comparison shows that A$ would come before B$ lexicographically. (In this case the lexicographical order is determined by means of a comparison of the ASCII codes of the characters – see Appendix H.)

Lexicographic order is basically the order in which words would appear in a dictionary, but including other types of character as well as letters. In particular, numbers come *before* letters, and most other symbols such as +, * and / come before numbers. Thus A+ comes before A1, which comes before AA. Note that a lexicographical comparison of numbers is quite different from a numeric comparison. "19" comes before "9".

Example 4.3

The next program uses this idea to sort two words into alphabetical order.

```
 10 REM ALPHANUMERIC SORT
 20 T$="Alphanumeric order is: "
 30 CLS
 40 INPUT "Type two words: "A$,B$
 50 PRINT T$
 60 REPEAT
 70   X$=A$
 80   A$=B$
 90   B$=X$
100 UNTIL A$<=B$
110 PRINT A$,B$
```

X$ is an extra string variable used to hold the contents of A$ temporarily during the swap.

Exercise 4.5

Modify Example 4.3 to sort three words instead of two by adding the following lines

```
 40 INPUT "Type three words: "A$,B$,C$
 65 REPEAT
110   X$=B$
120   B$=C$
130   C$=X$
140 UNTIL B$<=C$ AND A$<=B$
150 PRINT A$,B$,C$
```

When you have done these modifications, LIST the program to see how it works. Try it with data that is already in the correct order, e.g. P,Q,R; in reverse order, e.g. Z,Y,X; and in mixed order, e.g. N,L,M.

4.4 STRING MANIPULATION FUNCTIONS

4.4.1 String analysis

BASIC contains three useful functions to break large strings down into smaller ones. These functions are of value when only part of a string is actually needed for a particular task or if extra characters need to be inserted into a string.

To make proper use of these three functions it is frequently necessary to know the number of characters in the original string. Since the programmer may not know the length of a string in advance, it is convenient to mention at this point a further function which counts the number of characters in a string. LEN(A$) returns the number of characters in the string A$. Try the following simple program which will print out the number of characters in the user's name.

Example 4.4

```
10 PRINT "Please type your name"
20 INPUT NAME$
30 NUM=LEN(NAME$)
40 PRINT "There are ";NUM;" characters in your name"
50 PRINT "Thank you ";NAME$
```

Two of the functions for breaking down strings are rather similar. LEFT$(A$,N) returns the first (leftmost) N characters of the string A$. RIGHT$(A$,N) returns the last (rightmost) N characters of A$. The functions are used when the characters at the beginning or end of the string are needed.

Example 4.5

A simple example of their use which identifies the first and last characters in the user's name is as follows

```
10 PRINT "Please type your name"
20 INPUT NAME$
30 FIRST$=LEFT$(NAME$,1)
40 LAST$=RIGHT$(NAME$,1)
50 PRINT "The first character in your name is ";FIRST$
60 PRINT "The last character in your name is ";LAST$
70 PRINT "Thank you ";NAME$
```

The third function will select *any* chosen part of a string. MID$(A$,N1,N2) returns N2 characters of the string A$, beginning with the N1th character. MID$ is a more versatile function than LEFT$ or RIGHT$ and will in fact do all that the others can do. For example, if A$="INTENDED", then MID$(A$,4,3) will return "END".

MID$ can also be used with only two parameters to obtain the right-hand end of a string. If N2 is omitted, MID$(A$,N1) will return all characters from the N1th position to the end of the string. This can sometimes be simpler to use than RIGHT$, which needs the length of the remaining string supplied. For example, with A$ as above, MID$(A$,4) and RIGHT$(A$,5) will both return "ENDED".

When strings have been subdivided with these functions it is frequently necessary to reassemble them or to add to them which can be done by simply adding strings together.

Example 4.6

This is a simple example of the use of MID$ is to search a string for the occurrence of a second string (as we shall see below, this facility is provided more directly by INSTR).

```
10 CLS
20 PRINT "Type the string to be processed"
30 INPUT A$
40 PRINT '"Type the string to be searched for"
```

```
 50 INPUT B$
 60 J=0
 70 REPEAT
 80 J=J+1
 90 IF MID$(A$,J,LEN(B$))=B$ THEN PRINT "String found at
    position: ";J
100 UNTIL J=LEN(A$)-LEN(B$)+1
```

Note the use of the apostrophe in a single PRINT statement instead of repeated PRINTs to create multiple new lines in this and the following programs.

Example 4.7

As a more extensive example of the use of MID$, try the following program, which provides a search and replace facility for a user-supplied string. This is a primitive version of the system used in word processors.

```
 10 CLS
 20 PRINT "Type the string to be processed"
 30 INPUT A$
 40 PRINT '"Type the string to be searched for"
 50 INPUT B$
 60 PRINT '"Type the replacement string"
 70 INPUT C$
 80 C=0
 90 L=LEN(A$)
100 DIM K(L)
110 FOR J=1 TO L-LEN(B$)+1
120 IF MID$(A$,J,LEN(B$))=B$ THEN C=C+1: K(C)=J
130 NEXT J
140 F$=A$: REM F$ WILL BE NEW STRING
150 FOR J=1 TO C
160 L=K(J)-1+LEN(F$)-LEN(A$)
170 L$="": IF L>0 THEN L$=LEFT$(F$,L)
180 L=LEN(A$)-LEN(B$)-K(J)+1
190 R$="": IF L>0 THEN R$=RIGHT$(F$,L)
200 F$=L$+C$+R$
210 NEXT J
220 PRINT '"There are ";C;" occurrences of ";B$;" in"
230 PRINT A$
240 IF C=0 THEN END
250 PRINT '"The new string is: "
260 PRINT F$
```

The actual search is carried out in line 130 as in the previous example. K(J) stores the positions of each occurrence and C counts their number. The new string is created at line 210, by concatenating the left and right hand parts of the original string, obtained by LEFT$ and RIGHT$, and the new string, C$.

Exercise 4.6

Write a program to input a string, divide it into two equal (or nearly equal) halves, and to insert the string " EXTRA " in between the two halves.

Exercise 4.7

Write a program to input a string and then to print it out with the order of the characters reversed.

BBC BASIC contains two further functions not commonly found, which extend the versatility of the BBC microcomputer for manipulating strings.

INSTR provides a direct search facility of the type carried out in Exercises 4.6 and 4.7 using MID$. INSTR(A$,B$) returns the position in the string A$ of the first occurrence of the string B$. If no match is found, INSTR will return a zero. If B$ is an empty string, INSTR will always return a 1.

An alternative form of INSTR is INSTR(A$,B$,N) which starts the search from the Nth character of A$. For example, with N=2 the search would omit the initial letter, which could be useful if, say, you were looking for a word which might or might not start with a capital letter.

Exercise 4.8

Try modifying the program of Example 4.7 to use INSTR by the following changes

```
 90 L=1
100 DIM K(LEN(A$)+1)
110 REPEAT
120 J=INSTR(A$,B$,L): C=C+1: K(C)=J: L=J+1
130 UNTIL J=0
135 C=C-1: REM REPEAT LOOP COUNTS ONE TOO MANY
```

Note that there is a bug in INSTR in BASIC I (documented in the BBC *User Guide*) whereby the stack is corrupted if B$ is longer than A$. The bug is actually worse than this, in fact. Where the form INSTR(A$,B$,L) is used, the bug occurs if the *remaining portion* of A$ from L onwards is smaller than B$ (even if the whole of A$ is longer than B$). The statement that 'the stack is corrupted' is likely to be meaningless to a beginner, and probably will not explain clearly even to the experienced programmer what may go wrong. The result can actually be a very subtle flaw in program execution. The program modification above *ought* to work in all cases, but in fact under BASIC I the program will crash if B$ occurs at the very end of A$ (for example, if searching for "END" in "DIVIDEND"). What happens is that, on the pass of the REPEAT loop following the one in which "END" is deleted (when L=6), we have L=7 and the INSTR only examines the portion "ND" of "DIVIDEND", which is shorter than B$. This ought to return a zero into the variable J, but because of the stack corruption this does not happen, and the REPEAT loop ploughs on until the program crashes. The bug has been cured in BASIC II.

Exercise 4.9

Modify Exercise 4.8 to cater for the BASIC I bug in INSTR.

The second extra string handling function in BBC BASIC is STRING$ (not to be confused with STR$), which will produce multiple copies of another string. Thus

```
X$=STRING$(N,A$)
```

is equivalent to

```
X$=""
FOR J=1 TO N
X$=X$+A$
NEXT J
```

Its main use is likely to be in producing decorative features, such as strings of stars. For example

```
PRINT STRING$(40,"*");
```

will produce a complete line of stars. Similarly

```
PRINT STRING$(20-INT(LEN(A$)/2)," ");A$
```

will print A$ centred on a (40 character) screen line.

4.4.2 String conversion functions

There are four other special functions which relate to strings or string variables. The first two are a complementary pair which enable the programmer to change from a number to a string and vice versa. STR$(expr) converts a number or numerical expression into a string, for example

```
TEN$=STR$(10) or NUM$=STR$(NUM)
```

VAL(A$) performs the reverse process, finding the numerical value of a string, as in

```
TEN=VAL("10") or NUM=VAL(NUM$)
```

It follows therefore that VAL(STR$(A))=A and STR$(VAL(A$))=A$ (if A$ represents a number).

Exercise 4.10

Try

```
A=VAL(STR$(45)) then PRINT A
```

The function VAL may not produce a sensible result if the string does not in fact represent a number. The action of VAL in this case is to take any

numbers in the string up to the first invalid character. If the first non-zero character is invalid VAL returns a value of zero. Thus

```
VAL("ABC") gives 0
VAL("000XYZ") gives 0
VAL("12 BUCKLE MY SHOE") gives 12
```

But note

```
VAL("1.5E3") gives 1500
```

Try PRINTing these functions.

Exercise 4.11

Write a program to input a real number and determine whether it is of fixed point or exponent form (turn it into a string with STR$ and use LEN and MID$ to scan each character for a letter E). If it is in fixed point, set a new variable to the fractional part of the number (the part to the right of the decimal point) and then print the new variable. (Hint: find the position of the decimal point, discard the characters to the left with MID$ or RIGHT$ and finally form the numerical value with VAL).

1. Note that it is possible to generate the hexadecimal form of the number (see Section 10.1) by using a tilde before the brackets after STR$, for example STR$~(30).
2. Note also that there is a minor quirk with STR$: unless you alter the value of @% (see Section 11.4), the variable is converted to a string to full machine accuracy, whereas when the variable is PRINTed, it is rounded by one digit, which takes care of most rounding errors. Printing the STR$ of the variable will include any such rounding errors, of which you will not normally be aware.

Exercise 4.12

Try the following

```
A=2.1/10: B=3.1/10: C=3.2/10
PRINT STR$(A),STR$(B),STR$(C)
```

Try also printing values in hexadecimal, for example

```
PRINT STR$~(30)
```

To understand the last two functions we must first examine the way in which characters are read from the computer keyboard and stored in memory. There is a widely used standard code called ASCII (American Standard Code for Information Interchange) which allocates unique codes to each keyboard character, and also other keys such as carriage return, escape, control

characters and so on. The full list of these codes is given in Appendix H. (Note that BASIC programs are not stored completely as ASCII codes, but in a more complicated fashion, with whole keywords stored as a single code, for instance.)

There are times when it is necessary to use ASCII codes to output certain characters, when interfacing the computer to some peripheral devices for example.

The function CHR$(N) produces the character whose ASCII code is N, and the function ASC(A$) returns the ASCII code of the first character of the string A$.

Exercise 4.13

For example, try

```
PRINT CHR$(34)
```
– outputs a quotation mark

```
PRINT CHR$(91)
```
and
```
PRINT CHR$(93)
```
– output opening and closing square brackets.

Try also printing the characters with ASCII codes 7, 8, 10 and 12.

The ASC function is less used, but acts as the complement of CHR$, so check that ASC(CHR$(45))=45 and CHR$(ASC("M"))=M. (But note CHR$(ASC("LETTER"))=L.)

Exercise 4.14

1. Write a program to input any character and print its ASCII code.
2. Write a program to input an ASCII code and print the corresponding character.
3. Write a program to tabulate every ASCII code from 33 to 255 and the corresponding character.

On the BBC microcomputer, an alternative method of PRINTing characters is provided. This is via the VDU command.

```
VDU M,N,...
```

is exactly equivalent to

```
PRINT CHR$(M);CHR$(N);...
```

(except that you cannot have commands of the form VDU M,N,"STRING").

Thus if you want to PRINT several ASCII characters, and nothing else, it would be quicker to use VDU. For example

```
VDU 7,7,7
```

would give a triple length bell ring.

VDU is of course normally used to issue special screen control codes. For

instance, VDU 12 clears the screen. So does PRINT CHR$(12), therefore. By the same token, where a VDU command needs extra parameters, as in the case of VDU 19, trying to PRINT CHR$(19) will hang the computer, apparently inexplicably, until a few keys have been pressed.

In Mode 7 it is necessary from within a program to use either VDU or CHR$ to generate the special effect codes such as for colour and double-height text.

4.5 STRING ARRAYS

String variables, like real and integer variables, may have subscripts – they can be arrays – with the same limitation on the number of subscripts and default subscript sizes as for other arrays.

String arrays are extremely useful in many circumstances. For instance, Exercise 4.7 could be accomplished by loading the string one character at a time into a string array, then printing the array backwards in a decrementing loop

```
FOR J=LEN(A$) TO 1 STEP -1: PRINT A$(J);: NEXT J
```

where A$ is the original string.

Exercise 4.15

Repeat Exercise 4.7 using a string array.

With large arrays of strings, memory size can easily become a problem. This will not be apparent when the string array is DIMensioned, since memory is only used when strings are actually loaded into the string variable array elements (apart from an overhead of 4 bytes per array element).

Example 4.8

Consider the following program which simply keeps adding an extra character to each element of the array A$ to form the next element.

```
10 MODE 7
20 DIM A$(250)
30 A$(0)=""
40 FOR N=1 TO 250
50 A$(N)=A$(N-1)+CHR$(N)
60 PRINT ;N;" STRINGS STORED"
70 NEXT N
80 PRINT "PROGRAM ENDED"
```

If you run this program, you will see that it fails with error message 'No room', after approximately 220 array elements have been written to. Since each array element has as many characters as its array index, this corresponds

to a total memory requirement of approximately $0.5 \times 220^2 = 24K$, which is indeed most of the available memory.

Example 4.9

The program below shows the usefulness of string arrays. It is a program to display any message you choose with an attractive border of stars, and works by setting up a two-dimensional string array with stars all round the edge and your message nicely positioned in the middle. It uses several of the string facilities we have been discussing.

Lines 130 to 200 show how easy it is to draw the border with string arrays, while the real heart of the program is line 450 which puts the message into the array, after sorting out all the centring problems, and line 500 which prints the display onto the screen.

```
 10 SCRHT=25: SCRLEN=40
 20 REM IF NOT IN MODE 7, ALTER APPROPRIATELY
 30 DIM A$(SCRHT-3,SCRLEN)
 40 DIM B$(SCRHT-5)
 50 CLS
 60 PRINT "Initializing array...."
 70 FOR J=1 TO SCRLEN
 80 FOR K=1 TO SCRHT-3
 90 A$(K,J)=" "
100 REM INITIALIZE ALL ARRAY ELEMENTS TO SPACES
110 NEXT K
120 NEXT J
130 FOR J=1 TO SCRLEN
140 A$(1,J)="*"
150 A$(SCRHT-3,J)="*"
160 NEXT J
170 FOR K=1 TO SCRHT-3
180 A$(K,1)="*"
190 A$(K,SCRLEN)="*"
200 NEXT K
210 REM GENERATE BORDER OF STARS
220 CLS
230 PRINT "Type in a message, RETURNing at the"
240 PRINT "end of each line (maximum ";SCRHT-6;" lines)."
250 PRINT "When finished, just type <RETURN>."
260 K=0
265 REPEAT
270 K=K+1
280 INPUT B$(K)
290 UNTIL B$(K)="" OR K>=SCRHT-5
295 IF B$(K)="" AND K>1 THEN K=K-1
300 CLS
310 IF K>INT((SCRHT-6)/2) THEN GOTO 380
320 FOR J=K TO 1 STEP -1
```

```
330 B$(2*J-1)=B$(J)
340 B$(2*J-2)=" "
350 NEXT J
360 K=2*K-1
370 REM PUT BLANK LINE BETWEEN EACH LINE OF MESSAGE
380 L1=INT((SCRHT-5-K)/2)
390 REM L1 IS NUMBER OF BLANK LINES ABOVE MESSAGE
400 FOR KK=1 TO K
410 L=LEN(B$(KK))
420 L2=INT((SCRLEN-2-L)/2)
430 REM L2 IS NUMBER OF SPACES TO LEFT OF MESSAGE LINE
440 FOR J=L2+1 TO L2+L
450 A$(KK+L1+1,J+1)=MID$(B$(KK),J-L2,1)
460 NEXT J
470 NEXT KK
480 FOR K=1 TO SCRHT-3
490 FOR J=1 TO SCRLEN
500 PRINT A$(K,J);
510 NEXT J
520 NEXT K
```

4.6 FURTHER STRING HANDLING

String handling is probably the most important and useful non-mathematical use of the computer. The following program is a very simple word processor, which takes a continuously typed passage of text and displays it without words being 'wrapped round'. (The version on the supplementary disc contains lines 30 to 190 and 460, which are not printed below, simply to give the user an option to run a prepared passage instead of typing one in.)

Example 4.10

```
 10 DIM T$(800)
 20 CLS
200 PRINT "Type a piece of text, ending with"'"<RETURN>"'
210 L=0
220 REPEAT
230 L=L+1
240 T$(L)=GET$
250 PRINT T$(L);
260 UNTIL T$(L)=CHR$(13)
280 PRINT '"Press any key to print"
290 PRINT "this text properly"'
300 A$=GET$
320 K=0
330 REPEAT
340 J=42
```

```
350 REPEAT
360 J=J-1
370 UNTIL T$(J+K)=" " OR T$(J+K)="" OR (T$(J+K)="-" AND J<41)
    OR J=0
380 IF J=0 THEN J=40: REM WORD TOO LONG TO FIT
390 FOR M=1 TO J-1
400 PRINT T$(M+K);
410 NEXT M
420 IF T$(J+K)<>" " THEN PRINT T$(J+K);
430 IF J<40 OR T$(40+K)=" " OR T$(40+K)="" THEN PRINT
440 K=J+K
450 UNTIL K>=L
```

In this program, lines 340 to 370 are the key lines. Forty-one characters from the string of text are searched starting from the right-hand end until a space or hyphen is found at which the string can sensibly be broken. Line 440 then sets a counter to the start of the next 41 characters to be searched.

Another very common requirement is a program to sort strings. For instance, a class list may be stored as DATA statements, to which marks may be added, and then the list may be required to be sorted into mark order. There are many ways of sorting, but a simple and reasonably efficient method is the bubble sort. This method makes a series of passes through the list to be sorted, comparing each pair of entries in turn and reversing them if necessary. This means that the largest number in the list rises like a bubble to the end of the list.

The program remembers which numbers have reached their right place at the end of the list, so that each pass has less to sort than the previous one. The sort is complete when no interchanges are made during a pass. (In the program below, this is detected on line 210 when FLAG has the value 0.)

Example 4.11

The program below, which is intended to demonstrate how a simple bubble sort works, sorts strings up to 3 characters long, and shows each step of the process.

```
 10 REM BUBBLE SORT
 20 CLS
 30 PRINT "DEMONSTRATION BUBBLE SORT": PRINT
 40 INPUT "How many words? "N
 50 NN=N
 60 DIM A$(N): REM IN THIS EXAMPLE A STRING ARRAY IS USED
 70 PRINT '"Please type the words to be sorted,"
 80 PRINT "one to a line"'
 90 FOR I=1 TO N
100 INPUT A$(I)
110 NEXT I
120 PRINT ': PROC_printwords(N,NN)
130 REM SORT THE LIST
```

```
 140 REPEAT
 150 FLAG=0: N=N-1
 160 FOR K=1 TO N
 170 REM COMPARE TWO WORDS
 180 IF A$(K)>A$(K+1) THEN PROC_swap(K)
 190 NEXT K
 200 PROC_printwords(N,NN)
 210 UNTIL FLAG=0
 220 PRINT '"The list is now sorted"
 230 END
1000 DEF PROC_printwords(N,NN)
1010 LOCAL I
1020 FOR I=1 TO NN
1030 PRINT A$(I);TAB(4*I);
1040 NEXT I
1050 PRINT 'TAB(4*N);
1060 FOR I=4*N TO 4*NN: PRINT "-";: NEXT I
1070 PRINT
1080 TIME=0: REPEAT: UNTIL TIME=500
1090 ENDPROC
2000 DEF PROC_swap(K)
2010 LOCAL X$
2020 X$=A$(K): A$(K)=A$(K+1): A$(K+1)=X$
2030 FLAG=1
2040 PRINT TAB(K*4-4);A$(K);TAB(K*4);A$(K+1)
2050 TIME=0: REPEAT: UNTIL TIME=300
2060 ENDPROC
```

Try altering this program to sort words of any length, without the delay involved in the demonstration subroutine, by deleting lines 120, 200, 2040, 2050 and 1000 to 1080 to remove the illustrative procedure, and adding

```
225 PRINT '': FOR I=1 TO NN: PRINT A$(I): NEXT I
```

to PRINT the sorted list. This will show the speed of the sorting process. The program can also be used to sort numbers, if A$() and X$ are replaced by A() and X throughout.

Example 4.12

The next program uses the same bubble sort to carry out the class list problem. Ten student names are stored as DATA. The user must assign marks to each, and then the program lists the students by mark order.

The actual sort is carried out in lines 90 to 180 of the program.

```
  5 CLS
 10 DIM ST$(10),MARK(10),X(2)
 15 PRINT "Enter the mark for each student"'
 20 FOR J=1 TO 10
```

```
 30 READ ST$(J)
 40 PRINT ST$(J);TAB(10);"Mark = ";
 50 INPUT MARK(J)
 60 NEXT J
 70 N=10
 80 NN=N
 90 REPEAT
100 FLAG=0
110 N=N-1
120 K=1
130 REM COMPARE TWO MARKS
140 REPEAT
150 IF MARK(K)<MARK(K+1) THEN PROC_swap(K)
160 K=K+1
170 UNTIL K>N
180 UNTIL FLAG=0
190 PRINT '"Student      Mark"'
200 FOR J=1 TO 10
210 PRINT ST$(J);TAB(15);MARK(J)
220 NEXT J
230 DATA Adams,Black,Bloggs,Brown,Green
240 DATA Johnson,Jones,Smith,Thompson,White
250 END
1000 DEF PROC_swap(K)
1010 LOCAL X$,X
1020 X$=ST$(K): X=MARK(K)
1030 ST$(K)=ST$(K+1): MARK(K)=MARK(K+1)
1040 ST$(K+1)=X$: MARK(K+1)=X
1050 FLAG=1
1060 ENDPROC
```

Chapter 5

MATHEMATICAL FUNCTIONS

5.1 STANDARD FUNCTIONS

All microcomputers have a library of built-in mathematical functions, but the BBC microcomputer has a particularly comprehensive set. RND will be dealt with separately, but the remainder are listed below, with their effects.

There are two important points to remember

1. In all cases the opening brackets of the parameter or argument must follow without a space in between (for example INT(X), not INT (X)).
2. Most of the mathematical functions are calculated by approximation methods. This means that the results can not be guaranteed to be exact; there are always liable to be rounding errors. Thus you must be careful how you handle the results; for instance, never test the result of a mathematical function for equality. Either test with > or <, or test that it lies within a small range around the desired value. For example, if you want to test whether SIN(X) is 0.5, use the test

```
IF SIN(X)>0.499999 AND SIN(X)<0.500001
```

INT(X)

Gives the integer less than or equal to X. Thus INT(5.9) gives 5 and INT(−2.3) gives −3.

If you want to round to the nearest integer instead of always rounding down, use INT(X+0.5).

X must lie within the range of integer variables, between −2147483645 and +2147483645, otherwise the function will fail with the error 'Too big'.

ABS(X)

Gives the magnitude of X, that is the value ignoring any minus sign.

SGN(X)

Gives the sign of X

```
X<0 SGN(X)=-1
X=0 SGN(X)=0
X>0 SGN(X)=1
```

Thus SGN(X)*ABS(X)=X.

SQR(X)

Gives the positive square root of X. X must be positive.

PI

Returns the value of π.

DEG(X) and RAD(Y)

DEG(X) gives the value in degrees of an angle X in radians (1 degree = π/180 radians). RAD(Y) performs the reverse function. Thus RAD(DEG(X))=X (within the rounding errors of the conversion functions).

SIN(X), COS(X) and TAN(X)

These functions give the trigonometric functions sine, cosine and tangent. X must be in radians (but if it is in degrees, use SIN(RAD(X))).

ASN(X), ACS(X) and ATN(X)

These functions give the inverse trigonometric functions arcsin, arccos and arctan. The result is in radians, but to get Y in degrees, use Y=DEG(ASN(X)). For ASN and ACS, X must be in the range $-1<X<1$.

LOG(X), LN(X) and EXP(X)

LOG gives the logarithm of X to base 10, LN(X) gives the log to base e (natural logarithm), and EXP(X) gives e to the power X. Thus LN(EXP(X))=X. The only inverse function for LOG(X) is the exponent form of a number (for example 3E5).

The ranges of X are

X>0 for LOG and LN
X<=88 for EXP

Outside these ranges, the explicit error messages 'Log range' and 'Exp range' occur.

EVAL(FUNC$)

This function attempts to treat the string FUNC$ as a mathematical expression and evaluate it.

The purpose of the function is to allow the user to input a mathematical expression, in order to plot a graph of the function for instance. The EVAL

function is used in Example 9.9 to evaluate user input that may be in either decimal or hexadecimal.

DIV and MOD

These operators yield respectively the whole number part and the remainder of the division of two integer numbers. (If used with non-integer numbers, the numbers are truncated *before* the division, so that only the integer part of the number is used.) For example

```
13 DIV 3 gives 4
13 MOD 3 gives 1
```

Clearly B%*(A% DIV B%)+A% MOD B% = A%

AND, OR and EOR

These will be dealt with later in this chapter, but are listed here for completeness. In contrast to some microcomputers, they carry out full four byte, bitwise operations. EOR means *exclusive or*, and is not commonly available on other microcomputers.

Exercise 5.1

Check all the predicted relationships by direct mode commands such as

```
PRINT 3*(13 DIV 3)+13 MOD 3
```

Exercise 5.2

Write a program to INPUT as a string a mathematical expression, EXP(−X*X) for example, and use EVAL to print a table for a series of values of your expression and its parameter(s).

Exercise 5.3

Write a program for the user to enter an amount of money in pence, separate it into pounds and pence using DIV and MOD and print out the result.

5.2 RANDOM NUMBERS

The use and application of random numbers is worth considering in a little more detail. The random numbers produced by computers are more properly pseudo-random numbers, generated from the remainders after one large prime number is divided by another. To see how this works out in practice, try the following program.

Example 5.1

```
10 INPUT "Type any value (1 to 13) for the seed: ",SEED
20 CONST=19
30 DVSR=13
```

```
40 FOR J=1 TO 15
50 SEED=(CONST*SEED) MOD DVSR
60 PRINT SEED
70 NEXT J
```

The choice of 13 for the divisor gives 12 possible numbers, which as you will see come fairly randomly. The sequence then starts to repeat. The 'seed' simply decides at which point you join the repeating sequence.

If you want numbers between 0 and 1, this can simply be achieved by dividing the seed by 13.

With much larger prime numbers than 19 and 13, large sequences of random numbers can be produced. Try for example 4637 in place of 19 and 2546887 in place of 13 and replace line 50 by

```
50 SEED=CONST*SEED-INT(CONST*SEED/DVSR)*DVSR
```

since MOD will no longer work for such large numbers.

There are two alternative possibilities that you might need when using a random number generator. On the one hand, when using a program repeatedly, or developing a new program, you might want to use the same sequence each time, so that you 'know' the answer or can check other factors in your program. On the other hand, you may want to be sure of getting a different sequence each time, as in the card game we shall look at later.

BBC BASIC has a random number generator called by the function RND(X). This function has a number of different modes, depending on the value of X supplied. (As so often, X should be an integer, and real numbers are truncated.) These are summarized as follows

X absent	Random integer number between $+2^{32}$ and -2^{32}
X>1	Random integer number between 1 and X
X=1	Random fractional number between 0 and 1
X=0	Repeats the last number generated by RND if it was RND(1)
X<0	Returns the value X and reseeds the generator with a seed based on X

This choice caters for almost all eventualities. An effectively infinite range of numbers can be produced by RND or RND(1), depending on whether you want integer or fractional numbers. (To alter the range of real numbers, say to 0 to 100, use 100*RND(1). RND(100) would give an integer between 1 and 100.)

A random value from a set of integers up to N can be produced by RND(N). Thus to simulate the throw of a dice, you could use RND(6).

To get a repeatable sequence of numbers, start your program with X=RND(−1) (or any negative value which takes your fancy). To get a guaranteed non-repeatable sequence needs just a little more effort, since the generator always starts with the same seed when first switched on. The easiest

solution is to reseed with a random negative number, which can quite simply be produced from TIME, which is busily ticking away 100 times a second.

Example 5.2

A simple application to simulate the throw of two dice is given in the program below.

```
10 CLS
20 X=RND(-TIME): REM RANDOMIZE THE START POINT
30 INPUT "Number of throws: ",N
40 PRINT
50 FOR J=1 TO N
60 T1=RND(6): T2=RND(6)
70 PRINT "Throws are ";T1;" and ";T2;". Total ";T1+T2
80 NEXT J
```

Example 5.3

A simple program to test the statistical randomness of a sequence of numbers is shown below. Using RND(1), the mean should be 0.5, and the standard deviation should be SQR(0.08333/(N−1)).

```
 10 CLS
 20 INPUT "How many numbers are to be tested? ",N
 30 X=RND(-TIME)
 40 SUM=0: SUMDEVSQ=0
 50 FOR J=1 TO N
 60 X=RND(1)
 70 SUM=SUM+X
 80 SUMDEVSQ=SUMDEVSQ+(0.5-X)*(0.5-X)
 90 NEXT J
100 PRINT '"Mean is ";SUM/N;" (nominal 0.5)"
110 PRINT '"Standard deviation is ";SQR(SUMDEVSQ/(N*(N-1)))'
    "(nominal ";SQR(0.08333/(N-1)); ")"
```

Example 5.4

Finally, the more elaborate program below shows how a realistic computer version of the card game pontoon can be written with the aid of RND. The key line is line 1010 where the procedure selects a card using X(N)=RND(13)

```
 10 CLS
 20 DATA ACE,TWO,THREE,FOUR,FIVE,SIX,SEVEN
 30 DATA EIGHT,NINE,TEN,JACK,QUEEN,KING
 40 X=RND(-TIME)
 50 DIM CARD$(13),SCORE(2),PONTOON(2),ACE(2),X(10),Y(10)
 60 FOR J=1 TO 13: READ CARD$(J): NEXT J
 70 WINNINGS=0
 80 PLAYER=1
```

```
 90 SCORE(PLAYER)=0: PONTOON(PLAYER)=0: ACE(PLAYER)=0
100 WHO$="Your": WHICH$=" first"
110 N=1: PROC_deal(1): PROC_result(WHO$,WHICH$,CARD$(X(N)),
    PONTOON(PLAYER),SCORE(PLAYER),ACE(PLAYER))
120 INPUT "Place your bet, in pence: ",BET
130 WHICH$=" second"
140 N=2: PROC_deal(2): PROC_result(WHO$,WHICH$,CARD$(X(N)),
    PONTOON(PLAYER),SCORE(PLAYER),ACE(PLAYER))
150 IF PONTOON(PLAYER)=0 THEN PROC_twist
160 IF SCORE(PLAYER)>21 AND PONTOON(PLAYER)=0 THEN PRINT
    "BUST": WIN=-1: GOTO 290
170 IF ACE(PLAYER)>0 AND SCORE(PLAYER)<=11 THEN
    SCORE(PLAYER)=SCORE(PLAYER)+10
180 IF N>=5 THEN PRINT "FIVE CARDS": SCORE(PLAYER)=40
190 PLAYER=2: SCORE(PLAYER)=0: PONTOON(PLAYER)=0:
    ACE(PLAYER)=0
200 WHO$="My": WHICH$=" first"
210 N=1: PROC_deal(1): PROC_result(WHO$,WHICH$,CARD$(X(N)),
    PONTOON(PLAYER),SCORE(PLAYER),ACE(PLAYER))
220 WHICH$=" second"
230 N=2: PROC_deal(2): PROC_result(WHO$,WHICH$,CARD$(X(N)),
    PONTOON(PLAYER),SCORE(PLAYER),ACE(PLAYER))
240 IF PONTOON(PLAYER)=0 THEN PROC_dealer
250 IF SCORE(PLAYER)>21 AND PONTOON(PLAYER)=0 THEN PRINT
    "BUST": WIN=1: GOTO 290
260 IF ACE(PLAYER)>0 AND SCORE(PLAYER)<=11 THEN
    SCORE(PLAYER)=SCORE(PLAYER)+10
270 IF N>=5 THEN PRINT "FIVE CARDS": SCORE(PLAYER)=40
280 IF SCORE(2)>=SCORE(1) THEN WIN = -1: ELSE WIN=1
290 IF WIN=-1 THEN PRINT "YOU LOSE": ELSE PRINT "YOU WIN"
300 WINNINGS=WINNINGS+BET*WIN
310 IF WINNINGS<0 THEN PRINT "You owe me ";ELSE PRINT "I owe
    you ";
320 PRINT ABS(WINNINGS);" pence"
330 INPUT "Would you like to play again (Y/N)",A$
340 IF A$="Y" THEN GOTO 80
350 IF WINNINGS<0 THEN PRINT "Please pass my winnings into
    the grill"
360 END
1000 DEF PROC_deal(N)
1010 X(N)=RND(13)
1020 IF X(N)=1 THEN ACE(PLAYER)=N
1030 Y(N)=X(N): IF Y(N)>10 THEN Y(N)=10
1040 SCORE(PLAYER)=SCORE(PLAYER)+Y(N)
1050 IF ACE(PLAYER)>0 AND N=2 AND SCORE(PLAYER)=11 THEN
     PONTOON(PLAYER)=1: SCORE(PLAYER)=30
1060 REM PONTOON
1070 ENDPROC
```

```
2000 DEF PROC_result(WHO$,WHICH$,CARD$,PONTOONFLAG,SCORE,
     ACEFLAG)
2010 PRINT 'WHO$;WHICH$;" card is: ";CARD$
2020 IF PONTOONFLAG>0 THEN PRINT "PONTOON": ENDPROC
2030 IF N>1 THEN PROC_score(SCORE,ACEFLAG)
2040 ENDPROC
3000 DEF PROC_score(SCORE,ACEFLAG)
3010 PRINT "Total: ";SCORE;
3020 IF ACEFLAG>0 THEN PRINT " or ";SCORE+10;
3030 PRINT
3040 ENDPROC
4000 DEF PROC_twist
4010 INPUT "Twist or stick (T/S)",A$
4020 IF A$="S" THEN ENDPROC
4030 N=N+1: PROC_deal(N)
4040 PRINT '"Your next card is: ",CARD$(X(N))
4050 IF SCORE(PLAYER)>21 THEN ENDPROC
4060 PROC_score(SCORE(PLAYER),ACE(PLAYER))
4070 PROC_twist
4080 ENDPROC
5000 DEF PROC_dealer
5010 IF SCORE(PLAYER)>17 OR N>=5 OR (ACE(PLAYER)>0 AND
     (SCORE(PLAYER)<12 AND SCORE(PLAYER)>9)) THEN ENDPROC
5020 PRINT "I will twist"
5030 N=N+1
5040 PROC_deal(N)
5050 PRINT '"My next card is: ", CARD$(X(N))
5060 IF SCORE(PLAYER)>21 THEN ENDPROC
5070 PROC_score(SCORE(PLAYER),ACE(PLAYER))
5080 PROC_dealer
5090 ENDPROC
```

Note the use of recursion in the procedures PROC_twist and PROC_dealer. Most of the procedures have parameters because this is good practice, but it does make the program slightly longer.

5.3 LOGICAL EXPRESSIONS

5.3.1 Booleans

When a conditional test is made in an IF...THEN statement, what is happening is that a relationship is being evaluated as either TRUE or FALSE. A relationship which can have just the two possible values TRUE or FALSE is called a *Boolean*. More accurately, the IF statement has the general form

```
IF <Boolean expression> THEN...
```

On the BBC microcomputer, the Boolean values TRUE and FALSE actually exist as pseudo-variables (they can be read, but not altered). Thus it is quite legal (though rather pointless) to write

```
IF TRUE THEN...
```

In this case the statements following THEN would always be obeyed. A more useful application is with REPEAT...UNTIL, where again a Boolean is evaluated.

```
REPEAT...UNTIL FALSE
```

will create an endless loop, which can sometimes be useful. A loop that is to be repeated until a certain condition is fulfilled midway through the loop could have the structure

```
FOUND=FALSE
REPEAT
  ...
  ...
  ...
IF <condition> THEN FOUND=TRUE
  ...
  ...
  ...
UNTIL FOUND
```

Just as with arithmetic, we can have Boolean expressions as well as simple Booleans. There is a complete algebra for Boolean expressions. We will study this in full after reviewing the types of conditional test that are possible.

5.3.2 Conditional tests

The normal form of Boolean is a relationship, which involves testing two quantities with a *relational operator*, such as > (greater than). The full list of possible operators is

>	greater than
<	less than
=	equals
<>	not equals
>=	greater than or equals
<=	less than or equals

The double symbols *must* be in the order shown (e.g. <>, not ><) and *must not* be separated by a space.

The computer will evaluate the relationship as TRUE or FALSE, and assign a numerical value to the outcome (−1 for TRUE or 0 for FALSE). A

relationship can thus also be used within a normal mathematical expression.

Note that the equals sign has two quite separate functions in BASIC. It is used in an assignment statement, which strictly has the form

```
LET <variable>=<expression>
```

The LET is almost invariably omitted, but it does serve to emphasize that this is an assignment.

The second use of the equals sign is in a relational test. The computer finds no ambiguity in this, since it will treat = as an assignment where it is logical to do so, and a relational test otherwise.

Actually, a test for equality is not always a good idea, as was pointed out earlier when discussing functions. *Never* test real variables for equality if either has been evaluated in an expression, rather than having a directly assigned value (as the control variable of a FOR loop, for example).

Exercise 5.4

Check the fact that relationships have numerical values by direct mode statements such as

```
PRINT (1=2)
PRINT 15*(7<>3)
PRINT 7+(9<=10)
```

The fact that BASIC treats TRUE and FALSE as numbers means that it is also possible to write IF statements of the form

```
IF A% THEN...
```

A% will evaluate as TRUE if it has any value other than zero. Thus the above is a shorter equivalent to

```
IF A%<>0 THEN...
```

(Beware, however, that IF NOT A%... is *not* equivalent to IF A%=0...)

Check out these statements by lines such as

```
IF 10 THEN PRINT "TRUE" ELSE PRINT "FALSE"
```

The fact that a relational expression takes a value can be utilized to advantage to produce more compact programs and avoid IF...THEN statements.

For example, suppose we want to set a bus fare to 10p, 50p or 25p depending on whether the age of the traveller is under 15, adult or over 65. The obvious way to do this is

```
100 FARE=10
110 IF AGE>=15 THEN FARE=50
120 IF AGE>=65 THEN FARE=25
```

However, since a relational expression is −1 if TRUE and 0 if FALSE, this can be written much more compactly (though at the expense of clarity) as

```
100 FARE=10-40*(AGE>=15)+25*(AGE>=65)
```

Exercise 5.5

Rewrite lines 170 and 260 of the pontoon program, Example 5.4, using the above method.

Relational operators work with strings and string variables as well as numbers. The meaning of = and <> is fairly obvious, and useful in situations such as YES/NO tests. For example

```
100 INPUT "Do you want to continue? ",YN$
110 IF YN$="YES" THEN PROC_continue ELSE IF YN$<>"NO" THEN
    GOTO 100
```

Testing for inequalities is also possible, however, and carries out an alphabetical (or more strictly, lexicographical) test, ordering strings as they would appear in a dictionary. Thus

"FRED" is less than "JIM"
"BET" is less than "BETTER"
"10" is less than "20" (but "20" is less than "3")

Actually, what is done is a successive comparison of the ASCII values of the characters until a difference (if any) is found.

5.3.3 Boolean operators

It is possible to *combine* relationships with special operators called logical or Boolean operators. The available Boolean operators are AND, OR, EOR and NOT.

AND and OR perform just the logical operations that you would expect. EOR is the Exclusive OR operator, and is less common. Thus

`A AND B` is TRUE if A and B are both TRUE
`A OR B` is TRUE if either A or B is TRUE
`A EOR B` is TRUE if either A or B *but not both* is TRUE

NOT is rather different, being a unary operator which reverses the value of a relationship. (A unary operator is one which operates on a single quantity; the others are binary operators, and act on two quantities.) Thus

NOT TRUE is FALSE
NOT FALSE is TRUE

5.3.4 Order of priority of operators

The rules for combining operators are the basis of Boolean algebra. The mathematical and Boolean operators are listed below in order of execution in a compound expression. Operators on the same line have equal priority, and where these are combined they are evaluated from the left.

```
()
NOT (and + - as unary operators, as in 7*-6)
^
* /
+ -
> < <> >= <= =
AND
OR EOR
```

As with simple conditional tests, the Boolean operators find their most obvious use in IF statements. A pair of tests might be combined with a Boolean operator, for instance

```
IF X<1 OR X>4 THEN PRINT "Error"
```

or

```
IF NOT(R$="Y" OR R$="N") THEN PRINT "Error"
```

Exercise 5.6

Check the following statements, by using them as the <condition> in IF <condition> THEN PRINT "TRUE" ELSE PRINT "FALSE"

```
NOT -1=0        NOT 0=-1

0 AND -1=0      -1 AND -1=-1

0 OR -1=-1      0 OR 0=0

0 EOR -1=-1     -1 EOR -1=0
```

Exercise 5.7

Try a number of assignments in direct mode such as

1. `X=NOT((2+5)>7)`
   `PRINT X` gives −1 (TRUE)
2. `Z=NOT((3+2)>=4)`
   `PRINT Z` gives 0 (FALSE)
3. `Y=6=3 AND (2<>3 OR 1<2)`
   `PRINT Y` gives 0 (FALSE)

Try example 3 again without the brackets, and then with brackets around 6=3 AND 2<>3.

Note that Y=6=3 makes perfectly good sense to the BBC microcomputer, since 6=3 is evaluated first as a relational expression.

The most useful application of Boolean expressions is within the IF...THEN statement, which tests the truth of a Boolean expression.

For example, examine line 5010 of the pontoon program, Example 5.4, where the computer decides whether to twist. Note also the use of brackets to ensure the desired logic.

Exercise 5.8

Extract line 5010 from Example 5.4 as a simple program

```
10 INPUT "Score ",SC
20 INPUT "Ace ",ACE
30 INPUT "N ",N
40 IF SC>17 OR N>=5 OR (ACE>0 AND (SC<12 AND SC>9)) THEN PRINT
   "STICK" ELSE PRINT "TWIST"
```

Run this using different values of Score, Ace and N to check the logic of the relationship.

5.3.5 Bitwise action of the Boolean operators

This last subsection assumes some familiarity with binary arithmetic. Readers who are not familiar with the binary number system may prefer to skip it.

Originally the Boolean operators referred only to single-bit quantities, but by convention they are now extended to apply to all the bits of a number, and the same operators are used to describe the process of carrying out the operation bit-by-bit on all the bits of two binary numbers of the same length.

On the BBC microcomputer, all the Boolean operators actually perform bitwise operations on 32-bit binary numbers (that is, the operators act on the full 32 bits of integer variables).

Thus 3 OR 6, which is written in binary as

```
011 OR 110
```

gives 111, or decimal 7.

127 AND 159, or in binary

```
01111111 AND 10011111
```

gives 00011111, or decimal 31.

21 EOR 24, or in binary

```
10101 EOR 11000
```

gives 01101, or decimal 13.

NOT gives a full 32-bit complement of a number, so

```
NOT 111 gives 111...11000
```

Readers familiar with the two's complement representation of signed numbers will recognize this as −8 (that is, NOT 7=−8).

This also explains the apparently curious use of −1, rather than 1, for TRUE. TRUE is NOT FALSE, and to 32 bits,

```
NOT 0=111...11111
```

which in two's complement is −1.

Exercise 5.9

Try evaluating some bitwise logical expressions, by printing in direct mode expressions such as those given above. For example

```
PRINT 3 AND 6

PRINT NOT 7
```

Chapter 6

GRAPHICS

6.1 INTRODUCTION

No book of this sort would be complete without a discussion of the superb graphics capabilities of the BBC microcomputer. This chapter presents the simpler aspects of using graphics. It is not possible to cover all the advanced features here, such as use of the more complicated PLOT commands, plotting over or under existing screen displays and special facilities for animation.

Nor shall we attempt to discuss graphics in Mode 7, which is entirely different from that in all the other modes, and mainly suitable for logos and the like.

6.1.1 The graphics modes

Setting aside Mode 7, the remaining modes can be subdivided first into graphics and non-graphics modes.

Mode 3 and Mode 6 give a two-colour, 25 line, text-only display of respectively 80 and 40 characters per line. They cannot be used for graphics. Graphics commands issued in these modes will have no effect, though they will not give rise to any error message that would not have arisen in Modes 0, 1, 2, 4 or 5. The reason why these modes cannot allow for graphics is that the displayed lines of text are separated by a small gap. This can clearly be seen if you use commands such as

```
MODE 6
VDU 19,0,4,0,0,0
```

Some users find this type of display more pleasing than the normal one, with a coloured rather than black background, and the lines separated by black gaps. Redefined characters can be used with these modes, however, and this is one reason why they use nearly as much memory as the graphics modes (the difference arises from the saving due to the smaller number of lines of pixels stored).

The remaining modes: 0, 1, 2, 4 and 5, all nominally give a 32 line text

display with no gaps between the lines. They vary both in the number of characters per line and the number of colours available. Modes 0 to 2 require 20 Kbytes of memory and are only available on Model B computers, while Modes 4 and 5 take up 10 Kbytes, and can thus be used on Model A computers also, although in this case they leave rather little memory available for the BASIC program.

Table 6.1 summarizes the properties of Modes 0 to 6.

Table 6.1 Properties of the display modes.

Mode	*Characters per line*	*Colours available*	*Memory used*	*Graphics*	*Resolution*
0	80	2	20K	YES	640×256
1	40	4	20K	YES	320×256
2	20	16	20K	YES	160×256
3	80	2	16K	NO	
4	40	2	10K	YES	320×256
5	20	4	10K	YES	160×256
6	40	2	8K	NO	

Note that the colours available in a mode include black and white; thus a 2 colour mode is normally white on a black background, with no other colours available.

6.1.2 Pixels

The fundamental principle of graphics display is that the screen is divided up into a large number of 'picture elements', or *pixels*. Each pixel can be lit up in any of the available colours. Thus, like a newspaper picture, a graphics display is made up of small dots. The size of these dots determines the *resolution* of the picture. Tiny dots give a high resolution picture containing fine detail, whereas large dots give a much coarser picture in which fine detail is lost. It is common to express the resolution by the horizontal and vertical size of the display, expressed in pixels. For example, on the BBC computer, the resolution of Mode 1 is 320×256.

Knowing the resolution of the computer we can work out the number of lines of text that are possible, in the following way. Text characters all occupy character 'cells' which are made up of 8 rows of 8 pixels, with the appropriate pixels being lit to generate the particular character being displayed (one row and column are always blank in order to leave a space around characters, except for lower case letters with descenders). The way in which each

character is formed can be seen in the Appendix of the *User Guide*. All graphics display modes are 256 pixels high which allows 32 lines of 8 pixel high text. The horizontal resolution varies from 160 to 640 pixels, depending on mode. The 40 character modes give a nearly square display, where the height and width of each pixel is nearly equal. In other modes, the non-square nature will result in horizontal and vertical plotted lines having different widths.

In order to plot a point on the screen the nearest pixel must be lit up or 'plotted'. Obviously, any pixel could in principle be specified by its horizontal and vertical position on the screen. However, the same position on the screen would then need to be specified differently in the different modes, because the size of the pixels is different. On the BBC computer, therefore, a pixel is specified by means of *screen coordinates* which are independent of the mode.

The display area is always 1280 units wide by 1024 units high, and positions on the screen are specified in terms of these units. The *origin* of the display is at the bottom left of the screen, just as with a normal graph, and this position has coordinates 0,0. The possible positions on the screen are thus 0 to 1279 horizontally, and 0 to 1023 vertically, as shown in Figure 6.1.

It will be apparent that a pixel has dimensions of several screen units, and this will vary in the different modes. Figure 6.2 shows the size of a pixel in terms of screen units for the different modes. When a point or line is being plotted, a pixel is lit up if the point or line passes through *any* of the screen coordinates within it.

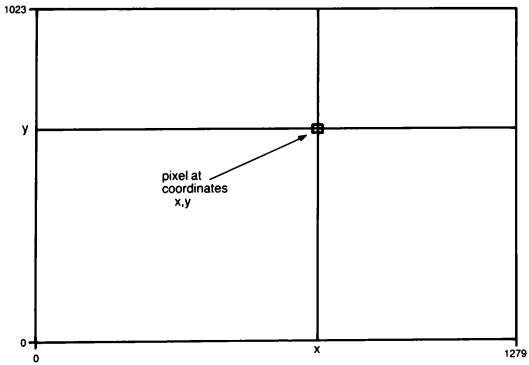


Figure 6.1 Coordinate system for the graphics modes.

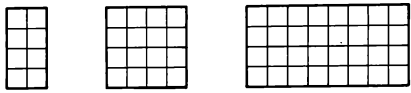


Figure 6.2 The size of a pixel in screen units for the different graphics modes.

The biggest problem with the limited resolution available on a microcomputer is not, as might be expected, the limitation on fine detail, although this is a consideration, but the unpleasant effect that can occur with straight lines. This particularly affects lines that are nearly horizontal or nearly vertical. With a slightly sloping line, the ideal line will pass successively through several pixels in the same row, after which it will cross the boundary between one row of pixels and the adjacent row, and abruptly the screen line will jump to that row for several more pixels. The result is that the sloping line becomes a jagged series of horizontal or vertical lines.

The effect is not confined to microcomputers, and you can observe the effect, for example, on television clocks. It is most easily seen on the second hand as it reaches one second to or one second past the minute.

Example 6.1

The following commands will demonstrate the resolution limitation very clearly.

```
MODE 1
MOVE 0,496: DRAW 1279,528
MOVE 0,0: DRAW 1279,1023
MOVE 628,1023: DRAW 656,0
MOVE 0,1023: DRAW 1279,0
```

Try these commands in other graphics modes as well, and notice the non-square effect showing up in the unequal line widths in Modes 0, 2 and 5.

6.1.3 Colour

The BBC computer provides an excellent range of facilities for colour displays and colour pictures of very high quality, particularly when the RGB output is used with an RGB colour monitor, rather than the UHF output and a television set. Text and graphics colours are set by separate commands, and both foreground and background colours may be controlled.

The number of colours available depends on the mode, and is shown in the table earlier. For text, the text colour can be changed by the command

```
COLOUR n
```

where n is the number for the desired colour. If n is greater than the range of colours available for the mode in use but less than 128, it is treated modulo that number. For example, in a four colour mode COLOUR 6 would actually select colour number 2, and in a two colour mode COLOUR 4 would select colour number 0.

For graphics the same rules apply, but the command is

```
GCOL 0,n
```

The first number specified can have a range of values in order to give rise to special effects, but in most cases it will be zero.

The colour numbers for the different modes are given in Table 6.2. In all cases the default colours for both text and graphics, set when the mode selection statement is executed, are white foreground and black background.

Table 6.2 Colours available in different modes.

Mode	*Colour*	*Colour number*	*Background number*
0,3,4,6	black	0	128
	white	1	129
1,5	black	0	128
	red	1	129
	yellow	2	130
	white	3	131
2	black	0	128
	red	1	129
	green	2	130
	yellow	3	131
	blue	4	132
	magenta	5	133
	cyan	6	134
	white	7	135
	flashing	8–15	136–143

The last eight 'colours' in Mode 2 are actually flashing combinations between pairs of complementary colours: colour n+8 flashes between n and 7−n. (The reason why only eight separate colours are provided is because these are the only eight colours that can be generated by various combinations of the three primary colours red, green and blue in equal intensity.)

Background colours may also be changed, and this is done by specifying the colour number plus 128 in the above commands. Thus to set a red text or graphics background in Mode 1 or 5, the command would be

```
COLOUR 129    or    GCOL 0,129
```

Note that this will not turn the whole screen red. It will simply provide a red background to any subsequent characters (including spaces) that are printed. The whole screen can be turned red by using CLS, which clears the screen to the current background colour. Similarly the graphics screen can be cleared to the current graphics background colour by CLG. Note that unless a text or graphics window has been set, these two ways of changing the background colour of the whole screen give the same result.

Exercise 6.1

Experiment with the COLOUR commands in direct mode, for different graphics modes, including changing the background. Note that if you make the foreground and background colours the same, subsequent text will become invisible.

6.2 PLOTTING LINES AND CURVES

6.2.1 The MOVE and DRAW commands

A great deal can be achieved in the way of graphics displays just by use of the two simple commands MOVE and DRAW, along with the GCOL command to set the graphics colour. The two commands can be envisaged as moving a pen across the screen to the position specified, starting from the position where it last came to rest. After a graphics mode is selected the pen starts from screen position 0,0. (Note that in OS 1.2 a CLG command does not restore the pen to 0,0, contrary to what is stated in the BBC *User Guide*.) The difference between the commands MOVE and DRAW is that with DRAW the pen leaves a trail of 'ink' in the foreground colour as it moves, whereas with MOVE it is 'lifted' from the screen and moves invisibly. The commands are followed by the horizontal (x) and vertical (y) screen coordinates to be moved to. Thus

```
MOVE 0,512
DRAW 1279,512
```

will draw a horizontal line through the middle of the screen.

Although the size of the screen is 1280 units by 1024 units, it is possible to plot outside this area, though obviously nothing will be seen. The plotting does exist, however, at least in the sense that a line drawn from somewhere off the screen to a position on the screen will appear to come from that hypothetical off-screen position. The overall limits on the x and y parameters to MOVE and DRAW are −32768 to 32767. (Larger values will not cause an error. It is a general feature of graphics commands, as well as other VDU commands, that they do not give rise to execution errors – they simply produce incorrect effects. For instance, numbers between 32768 and 65535 will be treated as negative numbers.)

Exercise 6.2

Experiment with these commands in direct mode. (Note that the text and graphics may superimpose on the screen, and if the text screen scrolls, it will carry the graphics display with it.) Try commands such as

```
MODE 1
GCOL 0,1
MOVE 0,512: DRAW 1279,512
MOVE 640,0: DRAW 640,1023
GCOL 0,2
MOVE 0,0: DRAW 1279,1023
MOVE 0,1023: DRAW 1279,0
```

Example 6.2

On some computers, there are commands to draw a line between a pair of specified points with a single instruction. Graphics is an ideal area for the application of procedures, and a procedure to draw a line from X1,Y1 to X2,Y2 in colour COL could take the form

```
1000 DEF PROC_line(X1,Y1,X2,Y2,COL)
1010 GCOL 0,COL
1020 MOVE X1,Y1
1030 DRAW X2,Y2
1040 ENDPROC
```

6.2.2 Plotting with straight lines

In view of the great variety of purposes for which simple graphics involving the drawing of straight lines and curves can be used, it is difficult to generalize how to tackle it, since different approaches will be needed for different problems. The most obvious division between uses is whether straight lines or curves are being plotted.

The most straightforward use of graphics is for plotting straight lines, since the DRAW command is directly appropriate for drawing any straight line; i.e. the software is already implemented. As an example, consider how to plot a monthly sales graph.

Example 6.3

```
  5 DIM SALES(12)
 10 REPEAT
 20 MODE 1
 30 PRINT "Input the monthly sales figures"
 40 PRINT "one per line, for January"
 50 PRINT "to December."
 60 MAX=0
 70 FOR J=1 TO 12
```

```
 80 INPUT SALES(J)
 90 IF SALES(J)>MAX THEN MAX=SALES(J)
100 REM MAX IS THE MAXIMUM SALES FIGURE
110 NEXT J
120 CLS
130 MULT=1: POW10=0
140 IF MAX<50 THEN GOTO 200
150 REPEAT
160 MAX=MAX/10
170 MULT=MULT*10: POW10=POW10+1
180 REM MULT IS MULTIPLE OF 10 SCALE FACTOR, I.E. 10^POW10
190 UNTIL MAX<50
200 SCALE=1-(MAX>10)-3*(MAX>20)
210 REM SCALE*MULT IS SCALE FACTOR PER DIVISION
220 GCOL 0,3
230 MOVE 128,148
240 FOR J=148 TO 948 STEP 80
250 MOVE 128,J: DRAW 1184,J: REM PLOT GRID
260 NEXT J
270 MOVE 128,144: DRAW 1184,144: REM PLOT ORIGIN DOUBLE
    THICKNESS
280 MOVE 124,144: DRAW 124,948
290 MOVE 128,148
300 FOR J=128 TO 1184 STEP 96
310 MOVE J,148: DRAW J,948
320 NEXT J
330 FOR J=0 TO 10 STEP 2
340 PRINT TAB(1,27-2.5*J);SPC(2-LEN(STR$(SCALE*J))); SCALE*J;
350 NEXT J
360 PRINT TAB(0,0);"x10^";POW10
370 PRINT TAB(3,28);"Jn Fb Mr Ap My Jn Jl Au Sp Oc Nv Dc"
380 PRINT
390 GCOL 0,1: PROC_pltpt(1,FN_ycoord(1,SCALE,MULT))
400 FOR J=2 TO 12
410 DRAW 32+J*96,FN_ycoord(J,SCALE,MULT)
420 PROC_pltpt(J,FN_ycoord(J,SCALE,MULT))
430 NEXT J
440 PRINT "Press SPACE for another graph"
450 PRINT "or any other key to end";
460 UNTIL GET$<>" "
470 END
2000 DEF PROC_pltpt(X,Y)
2010 LOCAL Y1
2020 FOR Y1=-8 TO 8
2030 MOVE 24+X*96,Y+Y1: DRAW 40+X*96,Y+Y1
2040 NEXT Y1
2050 MOVE 32+X*96,Y: REM MOVE TO CENTRE OF POINT READY TO DRAW
     LINE
```

```
2060 ENDPROC
3000 DEF FN_ycoord(J,SCALE,MULT)
3010 =148+80*SALES(J)/(SCALE*MULT)
```

This program plots a suitable sales graph for a set of monthly figures typed in by the user.

From the point of view of the graphics commands required, the only problem in the above program is in scaling the graph, since the screen is fixed at 1024 units high by 1280 units wide. If we are plotting the graph for one year, then a convenient horizontal scale would be 96 units per month (which corresponds to exactly three characters in Mode 1 or 4), leaving 128 units blank on the sides of the graph.

The choice of the vertical scale is not so simple. If the chart is to be split up into 10 divisions, it would be convenient to use 80 units per division, with 224 units (56 pixels or 7 lines of characters) shared between the spaces at the top and bottom.

After this choice has been made it will be necessary to find the largest monthly sales figure, and use this to determine the vertical scale for each of the ten divisions. If we want each division to represent a round quantity (normally 1, 2 or 5 times some power of 10) it is also necessary to make the scale larger still to accommodate the next higher round value.

Line 90 determines MAX, the maximum value to be plotted, which is used to calculate the vertical scaling. Lines 150 to 190 ascertain the power of ten needed in the scaling factor, the whole factor being stored in the variable MULT and the power in POW10. Line 200 determines whether each division will represent 1, 2 or 5 times MULT, using the fact that when evaluating a Boolean test, TRUE is −1 and FALSE is zero.

Lines 340, 360 and 370 print the scale and months on the graph, using the TAB command as described in Chapter 11.

Finally and most importantly, lines 250 and 310 respectively draw the horizontal and vertical grid lines, PROC_pltpt at line 2000 plots each sales 'point' and line 410 draws the lines joining the points, calling FN_ycoord at line 3000 to calculate the screen y-coordinate from the sales values SALES(J).

6.2.3 Plotting curves

For the most accurate representation, curves should be built up by plotting pixel by pixel, but this is generally too slow to be practical. Curves are almost always acceptable if plotted as short straight lines, but trial and error may be necessary to determine how long the straight lines can be.

The only curves which can be drawn simply from within a program are those with some definite mathematical formula. The simplest way to draw a curve is for functions that have the form

$$y=f(x)$$

which allows the curve to be built up by stepping along the x-axis, calculating y from the function at every step.

A simple example of plotting a sine wave is shown in Example 6.4. Try different values of the order n of the curve (which is of the form sin(nx)) and the step length (step length here means the horizontal length of each line *in pixels*) to see what step length gives an acceptable compromise between resolution of the curve and speed of execution for different orders. Note that some interesting effects can be obtained with the program below if very large values are specified for n.

Example 6.4

```
 10 MODE 1
 15 REPEAT
 20 PRINT TAB(0,26); "What order of sine curve would you"
 30 INPUT "like? (1-10) "N
 40 PRINT "What step length would you like?"
 50 INPUT "(1-50) "STP
 60 GCOL 0,3
 70 X0=640: Y0=608
 80 MOVE 0,Y0: DRAW 1279,Y0
 90 MOVE X0,192: DRAW X0,1023
100 REM PLOT AXES
110 XMAX=INT(636/STP)*STP
120 YMAX=400
130 MOVE X0-XMAX,Y0+YMAX*SIN(-N*PI)
140 FOR J=-XMAX+4*STP TO XMAX STEP 4*STP
150 X=PI*J/XMAX: REM SCALE ANGLES
160 Y=YMAX*SIN(N*X)
170 DRAW J+X0,Y+Y0
180 NEXT J
190 PRINT "Press SPACE to plot another curve"
200 PRINT "or any other key to end ";
210 UNTIL GET$<>" "
```

The variable STP specifies the step length, and is multiplied by 4 (since the program is working in Mode 1) so that it directly represents the number of pixels. Remember in your own applications that this step length always needs to be an integral number of pixels, otherwise you may get uneven results in your curves.

For sine curves scaling is simple since sin(nx) has a maximum value of 1, so y values are simply multiplied by YMAX (400). Similarly, the maximum value of angle wanted is π, so the x values can be scaled fairly simply. The X0 and Y0 added to the x and y values shift the origin to the centre of the top section of the screen (leaving room for a few lines of text at the bottom). Note that there is an alternative way of moving the origin to the centre of the screen, which is discussed in Section 6.3.2.

The next example plots a parabola, with a variable position for the focus, which means that the function must be scaled horizontally so that the curve

makes the maximum use of the screen area. This is done by the scale factor SCALE, calculated in line 140 and used to scale X*X in line 170.

Example 6.5

```
 10 MODE 1
 15 REPEAT
 20 PRINT TAB(0,26);"What focal length would you like?"
 30 INPUT "(12-400) "A
 40 PRINT "What step length would you like?"
 50 INPUT "(1-50) "STP
 60 GCOL 0,3
 70 XO=640: YO=192
 80 MOVE XO-12,YO+2*A: DRAW XO+12,YO+2*A
 90 MOVE XO,YO+2*A-12: DRAW XO,YO+2*A+12
100 REM PLOT FOCUS
110 XMAX=INT(636/STP)*STP
120 MOVE XO-XMAX,YO: DRAW XO+XMAX,YO
130 REM PLOT BASE LINE
140 SCALE=(800-A)*4*A/(XMAX*XMAX)
150 YOLD=A
160 FOR X=4*STP TO XMAX STEP 4*STP
170 Y=A+SCALE*X*X/(4*A)
180 REM FORMULA FOR A PARABOLA IS X*X = 4A(Y-A)
190 MOVE X-4*STP+XO,YOLD+YO: DRAW X+XO,Y+YO
200 MOVE -X+4*STP+XO,YOLD+YO: DRAW -X+XO,Y+YO
210 YOLD=Y
220 NEXT X
230 PRINT "Press SPACE to plot another curve"
240 PRINT "or any other key to end ";
250 UNTIL GET$<>" "
```

Plotting a curve as a series of straight lines for equal steps of X is not always the best method. One occasion when this is not the optimum strategy is when the curve has a vertical section. Compare the next two programs, both of which plot a circle of variable radius. The first uses equal steps of X, whereas the second program plots around the circle in equal steps of angle, which is much more satisfactory in the case of a geometric form such as a circle. The circle drawn on the screen may in practice be slightly elliptical due to variation in the height and width gain settings on your TV or monitor.

Example 6.6

```
 10 MODE 1
 20 INPUT "Radius of circle? (4-400) "R
 30 STP=INT(R/20): IF STP<1 THEN STP=1
 40 XO=640: YO=512
 50 MOVE XO-R,YO
```

```
 60 FOR X=-R TO R STEP STP
 70 Y=SQR(R*R-X*X)
 80 DRAW X+XO,Y+YO
 90 NEXT X
100 REM FORMULA FOR A CIRCLE IS X*X+Y*Y=R*R
110 FOR X=R TO -R STEP -STP
120 Y=SQR(R*R-X*X)
130 DRAW X+XO,-Y+YO
140 NEXT X
150 DRAW XO-R,YO
```

Here the number of steps has automatically been set to about 40, which is a reasonable compromise between speed of drawing and smoothness of the circle drawn.

Example 6.7

```
10 MODE 1
20 INPUT "Radius of circle? (4-400) "R
30 XO=640: YO=512
40 MOVE XO-R,YO
50 FOR ANGLE=0 TO 360 STEP 5
60 Y=R*SIN(RAD(ANGLE))
70 X=-R*COS(RAD(ANGLE))
80 DRAW X+XO,Y+YO
90 NEXT ANGLE
```

This time the step length is 5 degrees, giving 36 steps in each semi-circle.

Exercise 6.3

Alter the program from Example 6.7 to produce an ellipse by the following changes

```
15 INPUT "Semi-major axis of ellipse? (4-636)"A
20 INPUT "Semi-minor axis of ellipse? (4-400)"B
40 MOVE XO-A,YO
55 R=1/SQR(SIN(RAD(ANGLE))^2/(B*B)+COS(RAD(ANGLE))^2/(A*A))
```

Exercise 6.4

Finally, the original circle program of Example 6.7 can also be modified to produce a spiral, as follows

```
20 R=0
50 FOR ANGLE=0 TO 3600 STEP 5
55 R=ANGLE/10
```

6.3 THE PLOT AND VDU GRAPHICS COMMANDS

6.3.1 The PLOT command

Although a great deal can be achieved with MOVE and DRAW, these are only the two simplest of a large range of plotting commands. The full range is provided by the command PLOT, which has the form

```
PLOT K,X,Y
```

X and Y are the screen coordinates just as with DRAW, but K can take a large number of possible values, each giving a different form of plotting action. The factor K itself can be subdivided into three components, L + M + N.

L chooses the pen colour, while M determines whether the pen movement is relative or absolute. The choice of colours is: L=0, none (i.e. pen-up movement); L=1, the graphics foreground colour; L=3, the graphics background colour (this is not exactly the same as pen up movement because it will effectively eradicate any line already present by merging it with the background); and L=2, the logical inverse colour. This last colour we shall not consider further.

M=0 gives relative plotting, while M=4 gives absolute plotting. The latter means that the X and Y values specify the actual x- and y-coordinates on the screen, whereas when X and Y are relative, the value of X and Y specify the distance to be moved from the present pen position. Care must be taken when using relative plotting, as each relative movement can contribute a cumulative error to the position of the pen.

The effect of different combinations of L and M are shown in Table 6.3.

Table 6.3 Plot codes for different plotting colours.

Plotting colour	*Relative movement (M=0)* L + M	*Absolute movement (M=4)* L + M
Move only	0	4
Current graphics foreground	1	5
Logical inverse	2	6
Current graphics background	3	7

It can be seen from this that the commands MOVE and DRAW are exactly equivalent to PLOT 4 and PLOT 5. They are given separate, alternative BASIC keywords which are easier to remember and use because

of their special importance, but the PLOT form can equally well be used if you prefer.

The last part of the K parameter to PLOT gives rise to further variants, each available in the 8 possible colour/movement combinations set by L and M. The values of N and their effect are summarized in Table 6.4.

Table 6.4 Effects of different N values in PLOT.

N	*Action*
0	Simple drawing
8	Omitting the last point for inverting actions
16	Drawing a dotted line
24	Combination of 8 and 16
64	Plotting a single point only
80	Triangle filling

N=8 and 24 are beyond the scope of this chapter. N=16 is quite useful, but note that the effect is to draw dots in specific screen columns, so the effect is slightly limited in usefulness, and will vary depending on the angle at which the line is drawn.

Using N=64 to plot a single point provides an alternative to the combination MOVE X,Y: DRAW X,Y. The simpler result is

```
PLOT 69,X,Y
```

The triangle filling option is very useful. It works on the point specified along with the previous two pen positions. This provides a way of very rapidly filling any desired area with solid colour.

Example 6.8

The most likely shape that you may want to fill is a rectangle. The following procedure will fill a rectangle in the current graphics foreground colour, between X1 and X2, and Y1 and Y2.

```
1000 DEF PROC_rectangle(X1,Y1,X2,Y2)
1010 MOVE X1,Y1
1020 MOVE X1,Y2
1030 PLOT 85,X2,Y1
1040 PLOT 85,X2,Y2
1050 ENDPROC
```

Example 6.9

By combining the triangle filling with the program to draw a circle, we can write a procedure to draw a solid circle of any radius and centre, in the

current graphics foreground colour. The procedure below will draw a solid circle of radius R, centred at X0,Y0.

```
1000 DEF PROC_circle(R,X0,Y0)
1010 LOCAL ANGLE
1020 MOVE X0+R,Y0
1030 FOR ANGLE=5 TO 360 STEP 5
1040 MOVE X0,Y0
1050 PLOT 85,X0+R*COS(RAD(ANGLE)),Y0+R*SIN(RAD(ANGLE))
1060 NEXT ANGLE
1070 ENDPROC
```

6.3.2 The VDU command

We have already met the VDU command as an alternative to CHR$, and indeed the command

```
VDU M,N,...
```

is exactly equivalent to

```
PRINT CHR$(M);CHR$(N);...
```

The most valuable use of the command is to control the screen display (or VDU display, hence the name of the command) by the issue of control characters. Indeed, in direct mode a third way of achieving the same result is to type CTRL characters. For instance, CTRL-V CTRL-A would select Mode 1, as would VDU 22,1 (or MODE 1). The reason for this redundancy is presumably simply to enable advanced users to achieve all their needs with the VDU command, while beginners have simpler, easy to remember commands for the functions that they will need. The VDU commands which have an alternative form are listed in Table 6.5, and will not be considered further.

Table 6.5 VDU commands and their equivalents.

VDU *number*	*Control code*	*Equivalent*
9	CTRL-I	<TAB>
12	CTRL-L	CLS
13	CTRL-M	<RETURN>
16	CTRL-P	CLG
17	CTRL-Q	COLOUR
18	CTRL-R	GCOL
22	CTRL-V	MODE
25	CTRL-Y	PLOT
27	CTRL-[	<ESCAPE>
31	CTRL-_	PRINT TAB( );
127	N/A	<DELETE>

Non-graphics VDU commands

A number of the VDU commands are really nothing directly to do with graphics – they apply equally to text. Some of these are standard control codes, such as CTRL-G (VDU 7) to beep the speaker. These are contained in Table 6.6.

Table 6.6 Non-graphics VDU commands.

VDU *number*	*Control code*	*Effect*
0	CTRL-@	Does nothing
1	CTRL-A	Send next character to printer only
2	CTRL-B	Enable printer
3	CTRL-C	Disable printer
6	CTRL-F	Enable VDU drivers
7	CTRL-G	Beep the speaker (bell)
8	CTRL-H	Backspace cursor (but not delete)
10	CTRL-J	Move cursor down one line (linefeed)
11	CTRL-K	Move cursor up one line
14	CTRL-N	Switch on page mode
15	CTRL-O	Switch off page mode
21	CTRL-U	Disable VDU drivers or delete current line
30	CTRL-^	Home text cursor to top left

VDU 1, 2, 3, 6, 14 and 21 are dealt with in Appendix A. The remainder are self-explanatory.

The graphics-based VDU commands

The remaining VDU commands are used to control the graphics in various ways, and some have quite complex usage that needs to be described in detail. The commands are summarized in Table 6.7.

Table 6.7 Graphics-based VDU commands.

VDU *number*	*Extra bytes*	*Effect*
4	0	Write text at text cursor
5	0	Write text at graphics cursor
19	5	Define logical colour
20	0	Restore default logical colours
23	9	Define new character
24	8	Set graphics window
26	0	Restore default windows
28	4	Set text window
29	4	Set new graphics origin

To understand the effects of VDU 4 and VDU 5 we must first define the meaning of the text and graphics cursors. The text cursor is the position at

which text is normally printed on the screen, and it is governed by the TAB command in a PRINT statement, as well as being moved along every time a character is printed. Its position can be determined by POS and VPOS. The graphics cursor, on the other hand, is the position of the graphics pen as we have called it up to now. It is moved by the MOVE, DRAW and PLOT commands.

The two cursors are quite separate, but it is possible, by the command VDU 5, to switch text so that it is printed at the graphics cursor position instead of the text cursor (the text cursor remains at its old position, but no longer controls the text output). The text must now be positioned by the MOVE command. This has two main effects: firstly the position of text can be more precisely defined, to within one pixel in any direction instead of the nearest character cell. Secondly the screen will no longer scroll if output reaches the bottom of the screen; instead the output wraps round to the top of the screen. (If several screenfuls of text are printed, this causes overwriting and the text becomes unreadable.) Minor effects are that text colour and position are governed by the graphics colour setting and window (if any), the flashing cursor disappears, and characters do not obliterate the 8×8 square of pixels that they overwrite, so that they can be superimposed on each other, or on graphs. VDU 4 restores the effect of the text cursor.

Example 6.10

The following small program will print lines of text on the screen in one and a half line spacing (lines should be less than 40 characters long). Use ESCAPE to end the program. You can alter the line spacing by changing the factor 48 on line 60.

```
 10 ON ERROR GOTO 500
 20 MODE 1
 30 VDU 5
 40 J=0
 50 REPEAT
 60 MOVE 0,1023-48*J
 70 INPUT ""A$
 80 J=J+1
 90 IF J>21 THEN J=0
100 UNTIL FALSE
500 VDU 4
```

VDU 29 alters the position of the graphics origin. Very often, you will wish to centre your graphs, not on the default of the bottom left corner, but elsewhere such as at the centre of the screen. In our previous examples, this was achieved by defining X0 and Y0 and adding these into all MOVE and DRAW commands, but it can be more convenient to move the origin instead. VDU 29 is the first of these commands we have met that requires extra parameters. The 29 must be followed by two further numbers or variables, the x and y screen coordinates of the position where you want the new origin.

To move the graphics origin to the centre of the screen, for example, the command would be

```
VDU 29,640;512;
```

Successive parameters to the VDU command are normally separated by commas, but note that the x- and y-coordinates for VDU 29 must be followed by *semi-colons*. This tells the computer that they are two-byte numbers (since they may be greater than 255; you can alternatively break them up into low and high bytes, but this is very tedious). The semi-colon can be used at any time to indicate that the preceding parameter is a two-byte number, and this technique is sometimes used to concatenate redundant zeros, such as those at the end of VDU 19. Be careful, if you are doing this, that you do not omit the final semi-colon, as VDU commands do not give rise to error messages but can nevertheless result in peculiar effects. In this case, the next character input or issued would be seized as the last byte of the VDU command.

Exercise 6.5

Modify the programs in Examples 6.4 to 6.7 to use VDU 29 instead of X0 and Y0 for the graphics origin.

If VDU 29 is issued more than once, the newly specified coordinates are always relative to the initial origin at the bottom left-hand corner, not to the previously set origin.

6.4 SPECIAL GRAPHICS EFFECTS

6.4.1 Windows

It is possible to set *windows* for both the text and graphics displays. Initially either may be put anywhere on the screen, but if a window is set then the text or graphics, as appropriate, is restricted to the area of the screen inside the window – it is as if the defined area were a window through which you can look at the display behind. In the case of text, the text is restricted to the window area and the top left-hand corner of the text window becomes the new text origin, but graphics might well be plotted partly outside the area of the graphics window.

Text windows

Strictly speaking, text windows are unrelated to graphics, but it is convenient to treat them here along with the graphics window, especially as the two will often be used in conjunction.

A text window is defined by specifying the minimum and maximum character positions, both horizontally and vertically, in which text may appear. Figure 6.3 shows an example of this, in which only the top centre portion of the screen is used for a text window.

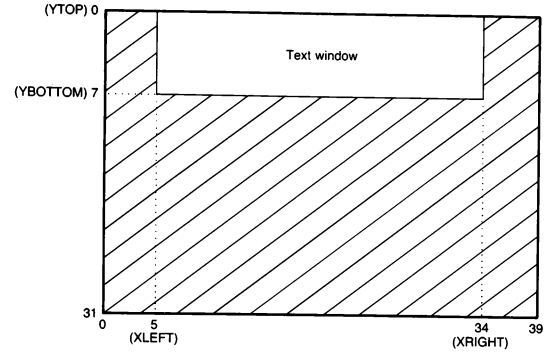


Figure 6.3 A text window as set up by the command VDU 28,5,7,34,0.

The command to set a text window is

```
VDU 28,XLEFT,YBOTTOM,XRIGHT,YTOP
```

You could think of this as specifying the bottom left and top right corners of the window. Note that for text Y is measured from the top, so YBOTTOM will be the *maximum* value of Y allowed. For the case shown in Figure 6.3, the command becomes

```
VDU 28,5,7,34,0
```

Note that the coordinates are the same as with the TAB command, and that the allowed ranges will vary with the mode in force. In this context you must remember that there are no error messages or error trapping for 'faulty' VDU statements – the command is simply ignored.

There are several reasons why you may wish to use a text window, quite apart from graphics windows. It would be the most convenient way of breaking program output into columns, particularly if the length of output may vary. A more important use is to keep a section of text permanently on the screen while other parts scroll. Once a text window is in force that does not encompass existing text, that text will not scroll, and so remains permanently on display. This technique is a favourite device in commercial programs such as word processors, where a top or bottom line showing status information remains permanently on display.

Any text window (or graphic window) may be cancelled with the command

```
VDU 26
```

The default full screen displays are also restored by a mode change (or, of course, a <BREAK>).

Example 6.11

The following program is a small demonstration which features the above points. It runs in Mode 6 to show that it is not restricted to graphics modes, but it could work equally well in any mode except Mode 2 or Mode 5, where the line length is too short.

```
10 MODE 6
20 PRINT "TEXT WINDOW DEMONSTRATION - THE ALPHABET"
30 VDU 28,5,9,34,2
40 FOR J=1 TO 26
50 FOR K=1 TO 240
60 PRINT CHR$(J+64);
70 NEXT K
80 PRINT
90 NEXT J
```

Exercise 6.6

Alter Example 6.11 so that it will work in Mode 2 or Mode 5.

Graphics windows

Graphics windows work in almost exactly the same way as text windows, except that, as with graphics displays on the full screen, the plotting range may run out of the window without ill effect, in which case only that part within the window is displayed.

The only difference is that the window will of course be in graphics screen units (with y measured from the bottom), which means that the x and y values will be greater than single bytes. They must therefore be two-byte parameters, and as before this is indicated by terminating them (even the last in the list) by semi-colons. The exact form of the command is therefore

```
VDU 24,XLEFT;YBOTTOM;XRIGHT;YTOP;
```

Note that this is the same convention as for the text window, specifying bottom left and top right corners of the window, but you must remember that this time YBOTTOM represents the smallest allowed value for Y.

The many conceivable uses include setting a window for a graph so that any lines that would otherwise spill outside the grid are suppressed. Another use could be to put a coloured border around a display.

A very useful application of windows is where text and graphics are to be used separately, but in the same display. Independent windows for the text

and graphics may be set so that they do not overlap, thus avoiding the possibility of one interfering with the other.

Example 6.12

The following program sets up a red graphics display area with a yellow border, and demonstrates the window effect further by attempting to draw a cross of full screen size.

```
10 MODE 1
20 GCOL 0,130
30 CLG
40 VDU 24,128;128;1151;895;
50 GCOL 0,129
60 CLG
70 MOVE 0,0: DRAW 1279,1023
80 MOVE 0,1023: DRAW 1279,0
```

Note that when no windows have been set, a background can be set to colour n either by

```
COLOUR 128+n: CLS
```

or

```
GCOL 0,128+n: CLG
```

but here CLG must be used because we wish to change the colour only within the graphics window. However, if text is subsequently printed onto the screen each character will have a black background unless COLOUR 128+n is also set, or VDU 5 is used to have text printed on a transparent background.

6.4.2 Changing colours

The limitation of most modes to 4 or 2 colours is not quite as restrictive as it may seem at first sight. Although the available colours in Mode 1, for instance, are by default black, red, yellow and white, it is possible to change these default colours. The command to do this is VDU 19. The change comes into effect immediately, changing the colour of pixels already displayed on the screen, and the command is frequently used to switch colours on and off to give an illusion of animation.

VDU 19 will change the colour that is associated with each of the permissible colour numbers in a particular mode. (In the *User Guide*, the colour number is called the *logical colour* and the colour associated with it the *actual colour*.)

The format of the command is

```
VDU 19,<colour number>,<actual colour>,0,0,0
```

The <actual colour> is specified by a number which is the same as the colour number sequence for Mode 2, as listed earlier. The trailing zeros are to allow for possible future expansion of the system.

Thus to change colour number 2 in Mode 1 from yellow to green the command would be

```
VDU 19,2,2,0,0,0
```

(or VDU 19,2,2;0; using the semi-colon to abbreviate the command).

The double occurrence of 2 in this particular example can be confusing, but a little thought will show that it is quite logical. If the colour number is outside the permitted range, then it is treated modulo the number of permissible colours, so in Mode 1 the same effect as above would be achieved by VDU 19,6,2,0,0,0. The command is not specifically a graphics command, and applies both to text and graphics colour.

Redefined colour numbers can all be set back to their default colours by

```
VDU 20
```

Exercise 6.7

Try changing the background colour of any of the earlier examples from black to blue, by using

```
VDU 19,0,4;0;
```

6.4.3 User-defined characters

One of the reasons why all the modes except Mode 7 take up so much memory is that each character is made up of an 8×8 pixel display. This means that it is possible to define new characters that will fit into an 8×8 display, or even redefine the existing character set. There is a command, VDU 23, that can be used to define characters.

Within the BBC microcomputer a character is specified by its ASCII code. The normal character set has ASCII codes from 32 to 127, with codes below 32 reserved for control codes (those that are normally issued by the VDU command). The default position for user-defined characters is that ASCII codes 224 to 255 can be defined by the user. The defined characters can then be printed either by PRINT CHR$(n) or VDU n. It is possible to change this default to a larger set, even to include redefining the standard characters, but extra memory has to be specially allocated for this purpose.

Defining a new character

A new character is defined by specifying which of the 64 pixels making up the 8×8 character cell are to be illuminated. Each row of the cell is defined by one number in the VDU string of parameters, made up in the following way: the right hand pixel counts as 1 if lit; the second as 2; the third as 4; and so on up to 128 for the left hand pixel. In this way all possible combinations of lit

and unlit pixels in the row can be represented by a single number between 0 and 255, which will just fit into a single byte of memory. (Readers familiar with bits and bytes will realize that each pixel is represented by one bit of the byte, the most significant bit representing the left-hand pixel.)

The 8 rows of the cell give 8 numbers, which are issued with the VDU command as follows

```
VDU 23,<char no>,a,b,c,d,e,f,g,h
```

The <char no> is the ASCII code of the character being defined, and a to h are the 8 numbers defining the rows of pixels, a representing the top row, b the second, down to h for the bottom row.

Let us now see how to define a new character, the greek character pi. First it is necessary to plan out the character on a piece of squared paper. Since it is to be a normal character, there must be a line of unlit pixels along the bottom and left hand sides, following the same convention used for normal characters so that they are separated from one another. Figure 6.4 shows the shape to be defined.

The values for the rows are calculated as follows: the top row has just the right hand pixel lit, so the value is 1. The second row has pixels 2 to 6 lit, so the value is 62 (32+16+8+4+2). The next row has a value of 84 (64+16+4), the following two have the same value of 20 (16+4) and row f, 18 (16+2). The seventh row has a value of 33 (32+1) and we must not forget the blank row at the bottom, which simply has a value of 0. If this is to be the first special character available, 224, the required VDU command is

```
VDU 23,224,1,62,84,20,20,18,33,0
```

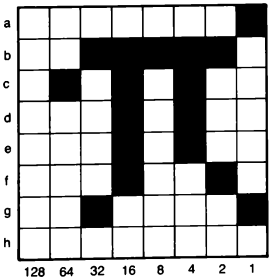


Figure 6.4 The shape needed for a user-defined character π.

Exercise 6.8

Try out the above character by the direct mode commands

```
VDU 23,224,1,62,84,20,20,18,33,0
MODE 6
PRINT CHR$(224);" = ";PI
```

Note that the character, once defined, is preserved through a mode change (or even a BREAK).

Try to define another greek letter, such as phi.

Exercise 6.9

Try and define some symbols for the sales chart of Example 6.3, in place of PROC_pltpt. Try shapes such as a square, a triangle and a circle, both solid and hollow.

User-defined characters are not restricted to single character cells. You can plan out a shape larger than 8×8 pixels, and spread it out over several character cells, which can be printed side by side or one above the other (using the TAB command). The sort of shapes that you might want are mathematical integral or summation signs, or small pictures. The latter will be plotted much faster in the form of user defined characters than by graphics commands. Each character can only be plotted in a single colour, but even this is not a major restriction, because the different colour components could be plotted as separate characters, superimposed with the aid of the VDU 5 command.

Making room for more user-defined characters

There is a *FX call, *FX 20,1, which allows the user to define as many characters as he wishes, or even redefine the normal character set. This uses up extra memory, at the start of the space normally used for programming. To allow for this, the normal value of PAGE for your system must be altered to accommodate the extra characters. (See Chapter 10 for a detailed discussion of PAGE, and general memory usage.) Provided that you have not already altered PAGE, the commands necessary to allow for extra characters are shown in Table 6.8 (note that the default character numbers are changed to 128–159 after *FX 20,1 – the call can also be used simply to make this change, if you wish).

Table 6.8 Making room for new characters.

ASCII character codes	*Command required*
128–159	none
160–191	`PAGE=PAGE+256`
192–223	`PAGE=PAGE+512`
224–255	`PAGE=PAGE+768`
32–63 (! to ?)	`PAGE=PAGE+1024`
64–95 (@ to _)	`PAGE=PAGE+1280`
96–126 (£ to ~)	`PAGE=PAGE+1536`

A full list of ASCII characters is given in Appendix H.

Switching the cursor off

There are two special purposes of the VDU 23 command. For the general user, the important one of these is to switch the flashing cursor off or on. Particularly when you have a graphics display, the flashing cursor can be an unwanted annoyance. It can be switched off with the command

```
VDU 23,1,0;0;0;0;
```

and switched on again by

```
VDU 23,1,1;0;0;0;
```

Note however that when text is linked to the graphics cursor by VDU 5, this also suppresses the flashing cursor, and if you are not wanting to print text at the text cursor position this is an easier way of removing it (VDU 4 brings it back, of course).

The second extra use of VDU 23, strictly for advanced users only, is to program the 6845 cathode ray tube controller chip (CRTC) which handles the screen display. We will mention only two possibilities here. First, the interlace which gives the screen display a slightly jittery effect, especially on some monitors, cannot be turned off in Mode 7 by the normal command *TV 0,1. It can be turned off, however, by

```
VDU 23,0,8,16,128;0;0;: VDU 23,0,9,9,0;0;0;
```

though you may think that the distorted character set that results is worse than the jitter.

The cursor can be altered in various ways by

```
VDU 23,0,10,n,0;0;0;
```

and

```
VDU 23,0,11,n,0;0;0;
```

In particular it can be turned off by

```
VDU 23,0,10,32,0;0;0;
```

and turned on again in its normal form by

```
VDU 23,0,10,114,0;0;0;
```

as an alternative to the VDU 23,1 commands. The above form will work even with the old 0.1 operating system.

Chapter 7

PROGRAMMING ERRORS

The errors that occur in programming divide broadly into four types

1. Syntax errors.
2. Semantic errors.
3. Execution or 'run-time' errors.
4. Logical errors.

The errors that are associated with the operating system and the disc errors are not discussed extensively. The operating system does not generate error messages since the operating system calls are not error trapped. This presents no problem to anyone who only uses the standard BASIC features, but the wide range of non-standard facilities are too valuable to omit from many programs. Once any of the operating system commands are used it is necessary to be exact and correct since no safety-net to trap errors is provided.

Example 7.1

The incorrectly written and poorly structured program fragment below does not give an error message when the incomplete VDU 23 statement that needs 9 arguments is executed. The arguments required are obtained from the following INPUT statement, which consequently gives an incorrect prompt!

```
10 MODE 4
20 PRINT "CHOOSE A LETTER"
30 VDU 23,234,129,255,129,
40 INPUT "NEVER GIVE A VOWEL",A$
50 IF (A$="A") OR (A$="E") OR (A$="I") OR (A$="O") OR (A$="U")
   THEN 20
60 PRINT "CHARACTER 234 ";CHR$(234)
70 PRINT "THIS IS THE END!"
80 END
```

The only protection against this type of error is careful and precise programming followed by thorough tests of all possible running conditions.

7.1 SYNTAX ERRORS

This type of error occurs whenever the computer cannot make sense of the commands that you have issued. There may be a variety of reasons for this. A typing error may result in an incorrectly spelt keyword which the computer will interpret as a variable. For example

```
10 IMPUT X
```

will generate the error message

```
Mistake at line 10
```

when it is run. This line may be inadvertently corrected to

```
10 IMPUT=X
```

if the spelling error is not noticed. This will generate the error message

```
No such variable at line 10
```

when it is run. The line may next be corrected to

```
10 INPUT=X
```

which, when it is run, will generate the error message

```
Syntax error at line 10
```

The correct form of the line is

```
10 INPUT X
```

It is important to realize that the error message needs to be interpreted as it may not be sufficiently specific to identify directly the error in the line. Any line containing an error should be read thoroughly to check that each character is correct. It should not be thought that all error messages are obscure; for instance the line

```
20 PTR=0
```

will generate the error message

```
Missing # at line 20
```

The error message is helpful if a program involving the use of the file pointer PTR#<channel number> is under development, but confusing if you want to

have a variable PTR to which you are assigning the value zero. However, once it is remembered that PTR# is a BASIC keyword and that these can not be used as variable names, it will be clear that PTR is not available as a variable name. It is, of course, perfectly in order to use ptr or Ptr as BASIC variable names.

Another way in which a syntax error may be generated is by an incorrectly inserted or omitted bracket which may confuse the computer. Another common error to watch out for is confusion between the letter O and zero. Apart from errors of typing you may have got the form of instructions confused, as for instance

```
FOR J=10 STEP 2 TO 20
```

These errors are usually easy to correct, since the computer always reports the line number in which the error occurred, so that this line can be LISTed and examined for faulty construction.

Certain syntax errors will generate specific error messages, such as

```
Missing ) at line 30
```

which would arise if the following lines formed part of a program which was then run.

```
10 DIM D(6)
20 FOR I=1 TO 6:NEXT
30 PRINT D(5
```

Another helpful syntax error message is 'Type mismatch' which will be generated by statements such as

```
10 A$=1
```

or

```
20 A="HELLO"
```

when a program containing them is run.

In the first case an attempt has been made to assign a numeric value to a string variable. In the second case an attempt to assign the string "HELLO" to a numeric variable has been made. Clearly the correction of errors such as these will depend upon what was originally intended.

The error 'No such variable' will occur if a variable appears on the right hand side of an equation or in a 'PRINT' statement before it has been given a value in a program statement such as

```
10 LATEST=0
```

Remember, however, that a statement in which a variable appears on both left and right hand side of the equals assignment such as

```
10 NUM=NUM+1
```

is allowed and will set NUM=1 the first time it is executed. The second and subsequent times that it is executed the value of NUM will be incremented as expected. The resident integer variables @% and A% to Z% may be used in PRINT statements and on the right hand side of equate statements without restriction and will have the values left over from use in previous programs. They will, however, need to be initialized if their previous values are not required.

An attempt to use an array element in a statement such as

```
10 Q(3)=7
```

without declaring the array by the statement

```
5 DIM Q(10)
```

will generate the error message

```
Array at line 10
```

But many syntax errors will just result in the message 'Syntax error' or even 'Mistake' if the computer cannot make head or tail of what you meant. It is then necessary to read the line of program concerned to check that intended statements have been correctly formed. If the cause of the error is not identified in the line at which it is reported, then the variables in the line should be traced in earlier lines of the program to check them.

7.2 SEMANTIC ERRORS

It is not always possible to make a hard and fast distinction between semantic and execution errors, but generally speaking semantic errors arise when commands which have correct syntax are incompatible in some way. For instance, if you accidentally cross a pair of FOR loops, or jump into a FOR loop, you will get the message

```
No FOR
```

or

```
Can't match FOR
```

Example 7.2

The program below contains a deliberate semantic error

```
10 DIM A(9)
20 I=0
30 GOTO 50
40 FOR I=1 TO 9
50 PRINT A(I)
60 NEXT
```

When it is run it will generate the message

```
No FOR at line 60
```

In this case line 60 is a valid line. The mistake has been made at line 30, which causes the program to jump into a FOR...NEXT loop. This mistake is clearly indicated by the error message. If line 20 is omitted the error message that occurs when the program is run is

```
No such variable at line 50
```

which is not so helpful. This is an example of an error message for which it is necessary to follow the program through to track down the cause.

The error message may not occur the first time a section of program is executed. The Example 7.3 contains crossed FOR...NEXT loops.

Example 7.3

```
10 FOR I=1 TO 10
20 FOR J=1 TO 10
30 PRINT I*J
40 NEXT
50 NEXT J
```

When this example is run on the computer it will generate the error message

```
Can't match FOR at line 50
```

after the first 10 numbers have been printed. This occurs because the BASIC interpreter has assumed that the NEXT at line 40 belongs with the FOR at line 20. The program can be corrected in two ways, either by changing line 50 to

```
50 NEXT
```

or by changing line 40 to

```
40 NEXT J,I
```

and omitting line 50. The latter saves a line of program.

In fact, all the messages of the form

```
No...
```

indicate semantic errors. Another example of a semantic error is

```
Arguments
```

which occurs when too many or too few arguments are given in a function or procedure call.

Most semantic error messages are self-explanatory. A list of all the possible error messages is given in Appendix J and the precise meaning of each is given in Section 46 of the BBC *User Guide*. The combination of an explicit error message with a line number makes semantic errors simple to track down in most cases.

7.3 EXECUTION ERRORS

These errors usually occur when the program itself makes complete sense to the computer, but, at least in certain circumstances, something goes wrong when the program is RUN.

In some cases these execution errors may be due to insufficient protection for unlikely data values which are not worth catering for, but very often they arise from a fault in the logic used to draw up the program.

Error messages usually associated with execution are relatively few, and would include

```
Accuracy lost
Bad DIM (this could also be due to a semantic error)
Division by zero
Eof
Exp range
Log range
-ve root
ON range
Out of DATA
String too long
Subscript
Too big
```

Errors such as 'No such variable' may also be errors of execution caused by the program taking an unplanned route and reaching a statement which uses the variable before it has been assigned.

By their very nature, execution errors are much the most difficult to locate, and may be due to the values which variables happen to have on one occasion. For instance

```
S=SQR(B*B-4*A*C)
```

is fine unless it happens that $4AC>B^2$, or B is greater than about 10^{20}, and the quadratic equation

$$Ax^2+Bx+C=0$$

has the perfectly good solution $x=-C/B$ when $A=0$, but use of the formula

$$(-B\pm S)/(2A)$$

will cause a 'Division by zero' error.

In association with the line number reported along with the error, an invaluable aid in tracking down these errors is the fact that all variables and array elements (including the control variables of FOR loops) retain their values after the execution of a program is halted due to an error. They are also retained if a program locked in an infinite loop is stopped by <ESCAPE> (but not <BREAK>).

When the source of an execution error is not obvious, therefore, the first thing to try, which will usually solve the problem, is to PRINT, in immediate mode, the variables on the offending line (and possibly some of the other program variables also, particularly the control variable of any FOR loop if the program stopped inside it). The values obtained should then be compared with those expected. Any discrepancy should be investigated as a possible source of error in the program. Alternatively an extra PRINT statement may be inserted in the program, inside the loop. This can be removed later when the error has been located and corrected.

7.4 RERUNNING THE PROGRAM

It may be that the values of the variables obtained, after the execution of a program has been stopped by an error, are themselves inexplicable. For instance, in the earlier example of a 'Division by zero' error, it might well be that the variable A should not have had the value zero because the program contained steps to cater for this eventuality.

The TRACE ON command causes the line numbers to be displayed as the program is executed each time a different line is reached. This enables you to trace the route taken through the program, and may for instance reveal that the A=0 trap was accidentally bypassed. TRACE ON can be inserted at any point in a program, to avoid an excessive listing of line numbers, and can also be switched off again with TRACE OFF, so you can just trace sections of your program if you wish. The TRACE ON facility is also turned off by the use of the <ESCAPE> and <BREAK> keys. There is a third form of the TRACE command, TRACE <line number>, which traces only lines with numbers greater than <line number>. This facility is provided in case you wish to avoid tracing through tried and tested procedures called earlier in the program.

It may be possible to correct a mistake in a program temporarily by reassigning variable values in immediate mode. It is then possible to continue execution by the command

```
GOTO <line number>
```

where <line number> is the faulty line which you have 'corrected'. Unlike RUN, GOTO does not clear the values of variables, so these are retained and the program may continue successfully. (This tip may even be of value in other circumstances, say where you missed reading some answers in the program before they rolled off the screen, or if you want to reprocess some numbers which took a long while to calculate or to type in.)

The STOP command can be useful in tracking down the cause of errors in a program. If it is inserted before a section of program that is suspected as the source of the error the values of variables can be checked, by PRINT statements in immediate mode, to see that they are as expected before this section is executed. The GOTO <line number> statement can then be used to show conclusively whether or not the error is in the section of program that is suspect.

Exercise 7.1

The program below which is available on the supplementary disc, contains a variety of errors of different sorts. RUN the program to track them down and try to correct them, and use the technique of PRINTing in immediate mode to help with any execution errors.

```
 10 REM INITIALIZATION OF VARIABLES
 20 DIM V(2)
 30 V(1)=70: V(2)=99: V(3)=140
 40 REM G=9.81
 50 REM DISTANCE OF TARGET
 60 D=RND(1500)
 70 REPEAT
 80 REM DISPLAY
 90 CLS
100 PRINT TAB(10);"Target at ";D;" metres"
110 PRINT 'TAB(14);"MAXIMUM RANGE"
120 PRINT '"Missile 1      Missile 2      Missile 3"
130 PRINT " 500m           1000m          2000m"
140 INPUT '"Select missile to fire (1,2,3) ",M
150 IF M<1 OR M>3 THEN GOTO 80
160 IF INT(M)<>M THEN GOTO 80
170 INPUT "Elevation (degrees) ",A
180 PRINT ''
190 REM SELECT SPEED OF MISSILE
200 V=V(-M)
210 REM CALCULATIONS
220 H=(V*SIN(RAD(A)))^2/(2*G: REM MAX. HEIGHT
230 T$=2*V*SIN(RAD(A))/G REM TIME OF FLIGHT
240 R=V^2*SIN(RAD(2*A))/G: REM RANGE
250 REM PRINTOUT
260 PRINT "Time of flight: ";INT(T);" sec"
270 PRINT "Maximum height: ";INT(H);" metres"
280 PRINT "Range          : ";INT(R);" metres"'
290 REM RESULT OF FIRE
300 E=INT(R)-D
310 IF ABS(E)<10 THEN GOTO 360
320 PRINT "Shot ";ABS(E);" metres ";
330 IF E>0 THEN PRINT "behind target" ELSE PRINT "short of
    target"
```

```
340 PRINT '"Press any key to continue: ";
350 A$=GET$
360 UNTIL ABS(E)<10
370 PRINT "DIRECT HIT! TARGET DESTROYED"
380 INPUT '"Would you like another go (Y/N)? "YN$
390 IF LEFT$(YN$,1)="Y" THEN GOTO 20
```

7.5 ERROR TRAPPING

In some programs it may be convenient to suppress error messages and to effect recovery from situations which would otherwise halt execution of the program. The command ON ERROR will accomplish this. It has four forms

```
ON ERROR GOTO <line number>
ON ERROR GOSUB <line number>
ON ERROR <statements>
ON ERROR OFF
```

The first three commands cause any subsequent error messages that would otherwise occur during the running of a program to be suppressed, and in the first two cases the program jumps to <line number> rather than having its execution stopped. <line number> would normally be the start of some error handling section. If GOSUB is used with ON ERROR then a RETURN must be executed at the end of the error handling section to allow the program to continue from the ON ERROR statement. There may be several ON ERROR statements in a program; the line number to which execution jumps in the case of an error is the one indicated by the last ON ERROR passed in normal execution of the program.

The last form, ON ERROR OFF, is used to cancel the error trapping, when you want normal error reporting to resume.

In Example 7.4, ON ERROR GOTO is used to trap two 'events'

1. The input of IMIN>IMAX which will lead to failure at line 110 in the square root function.
2. At line 180 a deliberate divide by zero is used to cause the request of the required YES or NO response.

In both cases the program will go to line 140. Try this example without line 10 first; this will enable you to find program mistakes rather than user mistakes.

Example 7.4

```
10 ON ERROR GOTO 140
20 REM TEST OF ON ERROR GOTO XX
30 CLS
40 PRINT "Fringe visibility calculation"'
60 INPUT "Imax and Imin ",I1,I2
70 V=(I1-I2)/(I1+I2)
```

```
 80 IF V<=0 THEN PRINT '"Imin must be less than Imax"
100 PRINT '"Fringe visibility = ";V
110 ROOT=SQR(V)
130 PRINT '"Root of V = ";ROOT
140 INPUT '"Do you want to do another calculation",A$
160 IF LEFT$(A$,1)="Y" THEN GOTO 60
170 IF LEFT$(A$,1)="N" THEN GOTO 190
180 V=V/0
190 END
```

Try the data 2,1 then 1,2 and responses other than YES or NO to line 140.

Exercise 7.2

Sometimes it is desirable to correct an error in some way and then allow the program to continue. This can be done if the GOSUB form of the ON ERROR statement is used.

Modify Example 7.4 as follows to illustrate the effects of ON ERROR GOSUB.

```
 10
 20 REM TEST OF ON ERROR GOSUB XX
 65 ON ERROR GOSUB 200: GOTO ERL
175 U=0
180 V=V/U
185 PRINT "RETURNed with U = ";U
188 GOTO 140
200 U=U+0.1: V=1: RETURN
```

Note that the RETURN is to the point immediately after the GOSUB on the ON ERROR line, rather than the point where the error occurs, which is what you would probably assume. This can be circumvented by the inclusion of GOTO ERL at the end of the ON ERROR line, which jumps back to the offending line. (Check that this is the case by leaving out GOTO ERL.) Actually this feature renders the use of GOSUB pointless – it would be simpler to use ON ERROR GOTO 200 and end line 200 with GOTO ERL instead of RETURN.

If the error is not corrected before RETURNing to the main program you can get into an infinite loop. Modify the program further by

```
200 U=U+0.1: PRINT "E";: RETURN
```

and use the data 1,2 in response to line 60. Press <BREAK> to get out of the infinite loop.

7.5.1 Error codes

When an error occurs in a program, the computer stores the code number of the error, and the line number on which it occurred, in the system variables

ERR and ERL respectively. In this way you can check in the error handling section which error occurred, and possibly deal with those you can anticipate and end the program with the rest. The error number of every error message is given in Appendix J, and you can look up the codes for the ones that you anticipate may occur.

For instance, the error numbers for '−ve root' and 'Division by zero' are 21 and 18 respectively, so we could further modify the program of Exercise 7.2 as follows

Exercise 7.3

```
200 U=U+0.1:V=1
210 IF ERR=18 OR ERR=21 THEN RETURN
220 PRINT "Error number ";ERR;" occurred at line number ";ERL:
    END
```

An interesting possiblility arises in that pressing the <ESCAPE> key is treated as generating an 'Escape' error with code number 17. Thus it is possible to protect your programs from users breaking out of them (whether by accident or on purpose) by trapping error number 17. To give further security you can also prevent users getting out of a program with <BREAK>, by issuing the command

```
*KEY10 OLD|M RUN|M
```

This gives limited protection unfortunately, as one can always get out of any program with <CTRL-BREAK>, which does not allow the <BREAK> key to behave as a function key.

Example 7.5

This program illustrates the redefinition of the <BREAK> key (function key 10) to provide limited protection for a program.

```
 5 ON ERROR GOTO 60
 6 *KEY10 OLD|M RUN|M
10 INPUT "Type a number from 1 to 10",A
20 X=1/A
25 X=SQR(A)
30 X=SQR(10.1-A)
40 PRINT "good, ";A;" is between 1 and 10"
50 GOTO 10
60 IF ERR=17 THEN PRINT: PRINT "Ha Ha, you can't ESCAPE that
   easily": GOTO 10
70 PRINT "Error number ";ERR;" occurred at line number ";ERL
80 GOTO 10
```

Try this program out with correct and incorrect responses, including letters, and try and get out of the program with <ESCAPE> and <BREAK>.

Finally, when reporting the error on line 70, the error number is not very illuminating. There is a command, REPORT, which will report what the last error to occur was.

Try adding to the above program the extra line

```
75 REPORT: PRINT
```

7.5.2 ON ERROR limitations

It is not possible to resume a program part of the way through a FOR... NEXT loop, a REPEAT...UNTIL loop, a procedure, function or subroutine. Instead, you will have to restart the loop, procedure or whatever. This is necessary since system pointers are changed by the operation of the ON ERROR GOSUB statement when an error occurs. If after an error an attempt is made to GOTO ERL to within a FOR loop, for instance, then a 'No FOR' error will occur.

Since this error will again be trapped with another jump into the error handling subroutine, the computer will get stuck in an infinite loop and the system will be effectively 'hung'. You will need to press <BREAK> or <CTRL-BREAK> to regain control.

Chapter 8

USE OF THE DISC FILING SYSTEM ON THE BBC MICROCOMPUTER

8.1 INTRODUCTION

The BBC microcomputer is capable of supporting a number of different filing systems for the interchange and storage of data and programs. All BBC microcomputers can support the cassette tape filing system. Additional firmware (ROMs) and hardware is necessary to support the floppy disc filing system, the Econet filing system, and the other filing systems available. The filing systems take care of the 'house-keeping' necessary to locate a file and allow the computer to transfer the contents of a file from the disc to the microcomputer. This takes only a few seconds, unlike the cassette filing system which may search a whole 30 minute tape until it either finds the file or runs out of tape.

In this chapter we are concerned with disc filing and it is assumed that a BBC microcomputer with an Acorn disc interface and at least one floppy disc drive is available. A dual floppy disc drive will be found convenient, and, in some applications, highly desirable. Some of the commands discussed here are common to all the filing systems whilst others are only available in disc systems. There are several other suppliers of disc interfaces and disc drives for the BBC microcomputer and in general these have common features with that of the Acorn system discussed here.

8.1.1 Discs and disc drives

In this type of storage system the recording medium is the magnetic coating of a rotating disc. Information is recorded along concentric tracks on the surface of the disc by a magnetic read-and-write head which can be positioned over any particular track. The high rotation speed and short head location time give access times which are very short compared with those provided by magnetic tape cassette. The favourable environment provided by the disc surface gives improved recording reliability. Each track is one bit wide and the data is recorded serially along the track.

The type of recording disc most frequently encountered on BBC microcomputer systems is a type known as a floppy disc, the name arising from the

flexible nature of the mylar backing material. The mechanism which drives and accesses the disc is known as a disc drive and the recording disc is sometimes referred to as a diskette.

The diameter of the disc used in systems controlled by the BBC disc filing system software is 5¼ inches. The disc is housed within a permanent stiff plastic casing (see Figure 8.1) which incorporates holes for read/write head access, an index mark and a write-enable slot which may be covered as a protection against any further writing to the disc. Discs must be inserted into the drive with the label on top and with the head access slot at the back of the drive. The disc is driven at the centre by a cone and plate which engages and aligns the disc when the flap or lever on the disc drive is placed in the closed position. When not in a disc drive, discs should be returned to their protective covering envelopes, as a small speck of dust or a greasy fingermark on the surface of the disc may cause an error ('Disc fault') when that track is read. This is similar to the good practice recommended for tape and video cassettes.

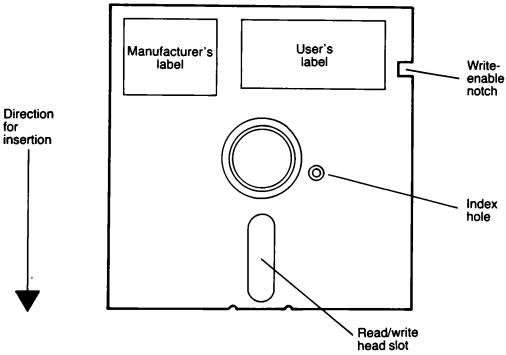


Figure 8.1 Features of a floppy disc.

The BBC disc system software permits the use of either 40 or 80 track disc drive units. The tracks are concentric circles spaced either 48 or 96 to the inch. The recording and reading head can moved to the appropriate track and has a total movement of slightly less than one inch to permit 40 or 80 tracks on a disc. Each track is divided into 10 sectors (see Figure 8.2). The Intel 8271 disc interface controller used in the BBC microcomputer only permits single density (as opposed to double density) recordings, and at the rotation speed of 300 RPM this determines a storage capacity of 256 bytes per sector. The storage capacity of a single-sided 40 track disc is therefore 10×256×40 bytes.

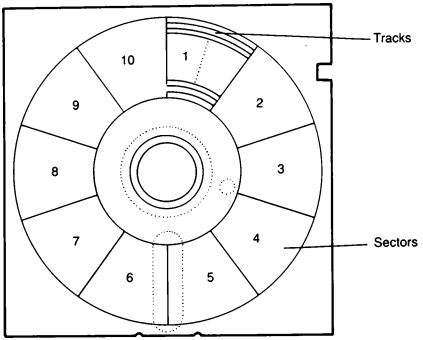


Figure 8.2 Organization of tracks and sectors on a floppy disc.

The term used for 1024 bytes is the kilobyte, so the corresponding storage capacity is 100K. A double-sided 80 track disc would have a storage capacity of 400K. Some disc drive units are switchable between 40 and 80 track operation. This is convenient when transferring files from one type of disc drive to another.

8.1.2 The disc filing system

Following power up a Model B microcomputer fitted with a Disc Filing System (DFS) ROM will normally display the message

```
BBC Computer 32K
ACORN DFS
BASIC
>
```

indicating the selection of the DFS as the default filing system. This message will also appear after <CTRL-BREAK>, and after <BREAK> the message is the same apart from the omission of the memory data (32K). Default conditions are those obtained when no specific action is taken to obtain other conditions. In the case of BBC microcomputers set up so that other filing systems (Econet for example) have default priority, the default conditions can be overridden to select the DFS by pressing <CTRL-D> and <BREAK> together. Likewise <CTRL-T>+<BREAK> will select the tape filing system. This can be verified on any machine, after a <CTRL-T>+ <BREAK>, by typing the *CAT command and observing that the

cassette motor light on the keyboard comes on. Pressing <BREAK> will normally return the machine to the default of the DFS.

Once a BBC microcomputer is switched on, any of the available filing systems fitted can be entered by typing the appropriate operating system command, which in the case of the DFS is *DISC or (for transatlantic cousins) *DISK, and can be abbreviated to *D. for convenience.

Two main versions of the DFS have been released by Acorn. The first was DFS 0.90 on an 8K EPROM and an enhanced version 1.20 is supplied on a 16K EPROM when a second processor is fitted. This later 16K EPROM also contains the Econet network filing system software (NFS) which would otherwise occupy a separate ROM socket. The most apparent difference is the more rapid file transfer under DFS 1.20. Data files are handled by DFS 1.20 in about half the time taken by DFS 0.90.

8.1.3 Getting started

Once the DFS has been selected, the BASIC (interpreter) commands SAVE and LOAD will be available for files on disc in the same way as they were for other filing systems. To find the names of files stored on a disc the command *CAT may be used (for a catalogue of the disc *CAT has the abbreviation *.). Any BASIC program may be loaded by

```
LOAD "<file>"
```

where <file> is a valid filename.

Any BASIC program stored in the computer can be copied to disc by the command

```
SAVE "<file>"
```

where <file> is the filename. (This command will overwrite any previous version of <file>.)

8.2 STRUCTURE OF INFORMATION ON A DISC

8.2.1 Files, directories and libraries

Information is stored on disc as a series of files, each identified by a unique filename. A file can be any program, data or other collection of information stored on the disc. Filenames are limited to not more than 7 characters or digits or both. Reserved characters such as * # . or : must not be included in the filename since these are used in special commands. The Acorn DFS does not distinguish between upper and lower case letters in filenames. All filenames and operating system commands can be in upper or lower case. The case used for a file name by the computer is that used when it is created, and either case may be used subsequently in commands given by the user.

Files are stored in sets known as directories, which are identified by any

single character except * # . : or a space. On power up or following a <CTRL-BREAK> the DFS automatically selects the $ directory (it defaults to the directory $): $ is said to be the *set directory*. It is not necessary to create a directory – they all exist but are empty until files are put in them. Directories on the DFS are really no more than prefix characters to file names, with $ having the special role of default directory and default library directory. The set directory can be changed to directory A (for instance) by use of the directory command

```
*DIR A
```

Following this, any commands to SAVE or LOAD programs will be assumed to refer to directory A (unless otherwise specified in the filename). For example

```
SAVE"PROG1"
```

will save the program as A.PROG1 and

```
LOAD"PROG2"
```

will load the program A.PROG2.

You might also have a program called PROG1 on the same disc in directory B. As this would be saved as B.PROG1 there is no risk of confusion. In general you may find it useful to store sets of programs or other files under different directories so that their common identity is recorded. This is aided by the fact that files belonging to the same directory are listed together in a *CAT listing.

In addition to the set directory, the DFS also allows for a *set library*. This is for use with the *RUN <file> or *<file> commands which will cause the specified program to be selected first from the current set directory if it exists, or if not, the currently set library will be searched for the specified file. The set library is one of the directories on the disc, the default setting being the $ directory. It is conventional to use the $ directory as the library directory as it is set by default. However, any other directory, for instance & which is conventionally chosen as the library directory on level 1 Econet, can be set as the library directory by

```
*LIB &
```

The advantage of having a selected library is that the *RUN command can be used to run programs from this library whilst some other directory is selected for all other purposes.

8.2.2 Dual and double-sided disc drives

A double-sided disc drive counts as two disc drives as far as the DFS is concerned. The selection of a drive, if more than one is fitted, is done by the command

```
*DRIVE n
```

where n is a number with the value 0, 1, 2 or 3 (n cannot be a variable, it must be a number). For example

```
*DRIVE 1
```

selects drive 1. Drives 2 and 3 correspond to the second sides of the discs in double-sided drives. The second side of the disc in drive 0 is called drive 2 and the second side of that in drive 1 is called drive 3. If a single double-sided drive is fitted the first side is drive 0, the second side is drive 2 and there is no drive 1.

The *CAT command has an extension which enables you to catalogue any drive, whether selected or not. This has the form *CAT n where n is one of the integers 0, 1, 2 or 3.

8.2.3 The catalogue of a disc

The current files on a disc can be obtained by the catalogue command

```
*CAT
```

(or *. in abbreviated form).

This will give a listing of the files to the screen (or currently selected output device) in the format shown in Figure 8.3.

```
*CAT
DISC-NAME  (34)
Drive 0              Option 0 (off)
Dir. :0.$            Lib. :0.$

    AFILE            BFILE
    CFILE L          ZFILE

  N.NEWFILE        Z.ANOTHER
  Z.ZFILE
```

Figure 8.3 *CAT display.

The *CAT command yields the following information: in the first line, the user-specified title for the disc – which may be blank – and in brackets the number of times the disc has been accessed for writing (modulo 100).

In the second line, the currently selected drive number is given (0, 1, 2 or 3) and the Option setting for the disc.

In the third line, the currently set directory and library are given in the form :0.$. The directory is $ and the 0 refers to the drive selected. The : and .

are *separators* that indicate that drive and directory parameters follow. The full specification for a file includes this information.

The next line of the display is blank and then there follows the current set directory filenames in two columns in alphanumeric order (by ASCII code values). At the end of the current set directory the files in other directories are arranged in a similar manner by alphanumeric order of directories, the files appearing in the form

```
N.NEWFILE      Z.ANOTHER
```

indicating the directory to which they belong.

8.2.4 Complete file specification.

In systems including more than one drive the file specification may include drive number information. For example the command

```
SAVE":1.K.PROGRAM"
```

would save the file PROGRAM in the K-directory of drive 1 even if the currently selected drive is number 0. This means that to load the program file AFILE from the disc catalogue shown above the following commands are all valid when the current set directory is $ and the drive selected is 0.

```
LOAD":0.$.AFILE"
LOAD"$.AFILE"
LOAD":0.AFILE"
LOAD"AFILE"
```

If another directory is the set directory then only the first two will load AFILE from the $ directory. The last two will load AFILE from the current set directory if it exists, or fail with the message 'Not found'. In multiple disc drive systems it is thus possible to load from or save to any directory on any disc drive by appropriate choice of parameters in front of the file name.

If the file exists in directory $ on disc drive 1 then it may be loaded by

```
LOAD":1.AFILE"
```

so long as $ is the current set directory, irrespective of the selected disc drive.

8.3 FORMATTING AND VERIFYING

The number of tracks on one surface of a disc is determined by the size and minimum movement of the read/write head which is driven by a stepper motor through a screw thread. The available aperture on the disc envelope permits a movement of the head of approximately one inch. There are two

common minimum step sizes: 1/48 of an inch and 1/96 of an inch, determined by the choice of stepper motor/screw thread combination, the precision of manufacture and the head which is much smaller for the smaller step size and must therefore be more accurately located on a track. The two step sizes are usually specified as 48 tracks per inch (TPI) or 96 TPI and floppy discs are often certified for these track densities. The disc drives which step 1/48 of an inch can accommodate 40 tracks and should be used with discs certified for 48 TPI. Disc drives with a step size of 1/96 of an inch can accommodate 80 tracks and need discs certified for 96 TPI. Discs certified for 96 TPI can be used for 40 tracks. Some disc drives can be switched from 80 to 40 track use – this is done by doubling the minimum track step size – but note that they use a small 80 track head and are not really suitable for writing to 40 track discs.

The rate at which data is read or written is determined by the floppy disc controller (an Intel 8271 disc controller chip in the case of the BBC microcomputer). It is necessary for the floppy disc controller to remain in synchronization with the data being transferred. In the case of the Intel 8271 this is done by interposing a clock pulse between each data pulse. These clock pulses are in addition to the synchronization provided by the single index hole punched in the disc. Discs which are synchronized in this way using a single hole, and in which the size of the sectors is determined by the formatting routine, are said to be soft-sectored. The hard-sectored 8" diameter discs have a second ring of holes for the start of each sector. The majority of 5¼" discs are soft-sectored, and only soft-sectored discs would normally be used with the BBC microcomputer.

The user establishes the sector structure on a soft-sectored disc by formatting the disc. This process is carried out by a program which uses special commands to the 8271, the BBC's floppy disc controller. During the formatting procedure, the appropriate synchronizing, address, data and checking bytes are placed on each track. The tracks are laid out in turn starting with the outermost which is track 0 and proceeding inwards. The tracks begin with a synchronization signal to identify the start of the track (in addition to the index hole). On each track, each sector begins with a synchronization signal to identify the beginning of a sector, and four bytes to identify the track number, side, sector number and sector size. The data bytes follow after further synchronization bytes. At the end of the data block a cyclic redundancy check, generated from the data, is stored for checking during loading.

The BBC microcomputer Model B+ has a formatter in the operating system which is absent from the Model B. However, all the suppliers of disc drives provide a formatting program. These differ in detail, but all carry out essentially the same task. We shall consider only the official Acorn/BBC formatter, which is used in essentialy the same manner whether from the utilities disc or from the operating system.

Assuming the use of a single-sided disc drive, the formatting process commences with the insertion of the utilities disc containing the formatter program (in the case of the Model B) and the command

```
*FORM40 0
```

is entered, which evokes the response

```
Disk formatter 1.05

Do you really want to format drive 0 ?
```

If the drive specification is omitted the response is

```
Disk formatter 1.05

Format which drive ?
```

to which the answer should be 0, which will then evoke the response

```
Do you really want to format drive 0 ?
```

as before.

Before answering this question the utilities disc is removed and replaced by the blank disc (on the Model B). The response Y then initiates the formatting process, during which, as each track is formatted it is also verified and the track numbers of completed tracks are displayed. The success message

```
Disk formatted
```

will appear when the process is successfully completed. If it fails the message

```
Verify error
```

or

```
Format error
```

will appear. At the termination of the process the option of formatting another blank disc is offered. The answer N then terminates the formatting program.

The verification of a disc or track checks the disc for 'readability' by reading each sector and checking that its checksum is correct. Verification forms an integral part of *FORM40 and *FORM80, and is also available as a separate utility program on the utilities disc as the file VERIFY and again is a built-in command on the Model B+. If this disc is placed in the selected library drive then

```
*VERIFY <drive>
```

will cause verification of the specified drive. This checks that all the files on the disc can be read, and can be used for reassurance about the condition of a disc.

8.4 OTHER DFS COMMANDS

In this section, round brackets are used to indicate an optional part of the command, and angle brackets, < >, are used for a general term such as the number of a disc drive. For example *CAT (<drive>) means that the following are valid

```
*CAT
*CAT 0
*CAT 1
*CAT 2
*CAT 3
```

In the case of *CAT the catalogue of files on the current set drive will be displayed, whilst in the other cases the catalogue of the disc in the specified drive will be displayed.

Wildcards are used for multiple file commands. A wildcard is a symbol which may represent any character (in the same way that the joker does in a pack of cards). The ambiguous file specification indicates that precise specification of files is not necessary. A group of files can be represented by the use of the * or # symbols, where # represents a single unspecified letter and * represents any number of unspecified letters or none at all. Thus ## can be used to represent any two letters. Partial specification can be used to select a group of files. For example

```
A.*
```

will select all files in directory A.

```
MY*
```

will select all files beginning with MY such as MYPROG,MY,MYKE whereas MY## will only select files of four letters beginning with MY.

```
*2*
```

will select all files containing 2 anywhere in the filename from first to last character.

The directory identifying letter can also be represented by a wildcard (either * or #) but the '.' cannot be so represented. The * and # only act as wildcards for valid filename and directory symbols. Thus to search all directories for files containing a 2 anywhere in the filename the wildcard sequence is *.*2* or #.*2*.

The full range of file handling commands of the DFS and of the Operating System are given in Table 8.1. There are in addition certain file handling commands associated with BASIC which are treated separately in Chapter 9. It is most convenient to consider the commands in groups.

Table 8.1 Filing System (DFS) commands.

Command	*Abrv.*	*Parameters*	*Comment*
*ACCESS	*A.	<afsp> (L)	
*BACKUP	*BAC.	<srcdrv> <destdrv>	Used for backup
*BUILD	*BU.	<fsp>	
*CAT	*.	(<drv>)	
*COMPACT	*COM.	(<drv>)	
*COPY	*COP.	<srcdrv> <destdrv>	
		<afsp>	
*DELETE	*DE.	<fsp>	
*DESTROY	*DES.	<afsp>	
*DIR	*DI.	(<dir>)	
*DRIVE	*DR.	<drv>	
*DUMP	*DU.	<fsp>	
*ENABLE	*EN.		Used with OS 0.90
*EXEC	*E.	<fsp>	
*HELP	*H.	(DFS) or (UTILS)	
*INFO	*I.	<afsp>	
*LIB		:(<drv>).<dir>	
*LIST		<fsp>	
*LOAD	*L.	<fsp>	
*OPT	*O.	M(,N) or M(N)	
*RENAME	*RE.	<oldfsp> <newfsp>	
*RUN	*R.	<fsp> (parameters)	
*SAVE	*S.	<fsp> <start> <end>	
		<exec> <load>	
*SPOOL	*SP.	<fsp>	Without <fsp> to end
*TITLE	*TI.	<disc-name>	Up to 12 characters
*TYPE	*TY.	<fsp>	
*WIPE	*W.	<afsp>	

fsp = file specification
afsp = ambiguous file specification
drv = drive number
dir = directory character

Round brackets are used to indicate optional parameters

8.4.1 Backup and file transfer

It is essential to be able to make copies of discs containing important programs and data. This can be done with the *BACKUP command, which has to be immediately preceded by *ENABLE (on DFS 0.90). This feature is provided to prevent the inadvertent overwriting of a disc which might otherwise occur during the backup sequence. To make an exact copy of a disc using a single disc drive (drive 0) the following sequence of commands is typed

```
*ENABLE
*BACKUP 0 0
```

The copying process is done in sections and requires frequent interchange of the source disc and the destination disc. It is advisable to cover the write enable notch on the source disc with a write-protect sticker so that the original disc cannot be overwritten accidentally. The new (destination) disc must have been formatted previously, and will be completely overwritten.

The process is much more straightforward when two disc drives are available, in which case the commands are

```
*ENABLE
*BACKUP 0 1
```

to copy the contents of the disc in drive 0, sector by sector, onto the disc in drive 1. It is usual to backup 40 track discs to 40 track discs, and 80 track to 80 track. It is, however, possible to backup 40 track discs to 80 track, but it is not possible to backup 80 track to 40 track discs.

The *ENABLE line can be omitted if DFS 1.20 is fitted since the response to

```
*BACKUP 0 0
```

is then

```
Copying from 0 to 0
Go (Y/N) ?
```

which is used to prevent unintentional backing-up. A Y response permits the backing-up to proceed, whilst N ends the effect of the command. The response to *BACKUP 0 0 on OS 0.90 is the error message

```
Not enabled
```

The *COPY command is available to transfer one or more individual files from one disc to another, and has the form

```
*COPY <source drive> <destination drive> <afsp>
```

For example to transfer a group of files containing the letters MY from a disc in drive 1 to a disc in drive 0 the form would be

```
*COPY 1 0 *.*MY*
```

The value of this command is that files can be copied onto discs that already have some programs on them without overwriting the existing programs and it can transfer between 80 and 40 track discs. The *COPY command has much to recommend it as it will not overwrite existing files, but since it transfers one file at a time it is slower than *BACKUP when a large number of separate files are to be moved from one disc to another. The *BACKUP command transfers all the information from one disc to another, ignoring the file structure, in as few stages as possible by using all the available memory on

the computer. Both *BACKUP and *COPY will overwrite existing programs in computer memory.

8.4.2 File removal

When discs have been in use for some time, files will accumulate that are no longer required. A single, named, file can be removed by the command

```
*DELETE <fsp>
```

A group of files can be removed by the commands

```
*ENABLE
*DESTROY <afsp>
```

Thus

```
*ENABLE
*DESTROY *.BWJ*
```

would give a list of all the files starting with BWJ and give the prompt

```
Go (Y/N) ?
```

to be answered Y to confirm that all the files are to be deleted. Again *ENABLE is not necessary with DFS 1.20 which gives an extra prompt after the *DESTROY command thus

```
*DESTROY
Go (Y/N) ?
```

Another command to allow selective deletion of a group of files is

```
*WIPE <afsp>
```

The name of each file corresponding to the ambiguous file specification is displayed, and the user replies Y if it is to be deleted, N if not. For example

```
*WIPE BWJ#
BWJ1    :Y
BWJ2    :Y
BWJ3    :N
BWJ4    :Y
```

will delete files BWJ1, BWJ2, BWJ4 but not BWJ3. The command *WIPE does not need *ENABLE because each file is separately presented for approval before deletion.

8.4.3 Disc organisation: *INFO, *OPT 1 and *COMPACT

The catalogue command *CAT which gives a list of the names of the files and their access status has been described above. Further information about the files on the disc can be obtained by use of the command

```
*INFO <afsp>
```

The information displayed for each file is: directory name, status, execution address, load address, length, sector address

For example

```
*INFO A.*

A.MENU L 001900 00801F 00003D 003
```

The numbers are in hexadecimal. This line shows that the file MENU is in directory A. The status L indicates that the file is locked to prevent accidental overwriting from occurring. The execution address is that normally found for a microcomputer fitted with DFS and the load address is the normal load address for a BASIC program.

This information can be made to appear each time there is any file access by use of *OPT command in the form

```
*OPT 1 1
```

The additional information can be suppressed by

```
*OPT 1 0
```

Although the DFS takes care of the storage location of each file and the catalogue of files on the disc, it does not always make the most efficient use of a disc. After files have been deleted, or when files are extended and then saved again under the same name, there will be gaps between one file and the next. A storage problem can occur when a file has others after it on the disc or when it is the last file and fills the disc. If this file is modified and if this makes it longer it may no longer be possible to save it back into the same space. This will cause the error message

```
Can't extend
```

or if there is no room on the disc

```
Disc full
```

These messages may be generated even when there are gaps between the files due to deleted files. In order to release this space for efficient use a special

command must be used which shifts files up on the disc so that no vacant sectors are left between one file and the next. This command is

```
*COMPACT (<drive>)
```

When this command is used, the program currently in memory may be lost, as the memory is used in the movement of files. It will therefore be necessary to save any program in memory to another disc with free space before use of the *COMPACT command. Files always use an integral number of sectors, and any space not used by a file at the end of the last sector is left blank.

8.4.4 File protection

The advantages of backing-up discs and copying files are that secure copies of files can be made. Limited security is available on the disc currently in use by means of the *ACCESS command which permits files to be 'locked' against being overwritten or deleted. For example

```
*ACCESS A.FRED L
```

will result in the file FRED being locked. Files that are locked are followed by an L in the catalogue listing obtained using *CAT. To unlock the file to amend it, *ACCESS without the L may be used. For example

```
*ACCESS A.FRED
```

The *ACCESS command can be issued for a group of files with the ambiguous file specification. For example

```
*ACCESS A.* L
```

locks all files in the A directory.

Once locked, a file is not affected by the DFS commands *SAVE, *DELETE, *DESTROY, *WIPE, *RENAME or the BASIC command SAVE "<file>". The *ACCESS command offers no protection to the *BACKUP and *FORMAT commands. The only protection against *BACKUP and *FORMAT is to use a write-protect label which protects the complete disc. The use of the write-protect label on the backup disc is recommended where valuable programs or data are stored on a disc. It should be noted that these precautions cannot protect a file from being physically corrupted, for example by dust or by a magnetic field (which might occur when a disc is left in the disc drive when it is switched on or off) and for these reasons separate backup copies of all important files should be kept.

8.4.5 Other DFS organization commands

*RENAME

The name of a file is usually chosen so that it reflects in some way the contents of the file. When new files are created it is sometimes found that there is a

conflict between the name of an existing file and that chosen for the new file. In these circumstances it is convenient to be able to change the name of a file; this can be done with the command

```
*RENAME <old fsp> <new fsp>
```

This command changes name of the file <old fsp> to <new fsp>.

***TITLE**

Each file on a disc has a name and they can be grouped into different directories. It can also be useful to give a name to the disc. This can be done by the command

```
*TITLE <disc name>
```

The command permits a title of up to 12 characters long to be appended to the disc in the currently selected drive. The title will then be displayed whenever that disc is catalogued.

***HELP**

There are occasions when it is useful to be able to find out certain characteristics of the particular microcomputer that you are using. This can be done with the command

```
*HELP
```

The command displays information concerning the ROMs fitted within the BBC microcomputer.

Other *HELP options

On other occasions a quick revision of the available commands can be useful, in which case the following modifications to *HELP will be found helpful

```
*HELP DFS
```

which dislays a list of the DFS commands with their syntax. Similarly

```
*HELP UTILS
```

displays the syntax of the DFS commands concerned mainly with the creation and listing of ASCII files.

If other ROMs are fitted in your computer, they may provide further *HELP options. For example, all the commands of the popular utility ROM *Disc Doctor* can be listed by *HELP DISC DOCTOR.

8.4.6 Machine code programs

The commands *SAVE, *LOAD and *RUN are associated with machine code programs. They are analogous to SAVE, LOAD and CHAIN and are discussed more fully in Section 10.8.5.

8.4.7 ASCII file commands

These commands are

```
*SPOOL <fsp>
*EXEC <fsp>
*BUILD <fsp>
*TYPE <fsp>
*LIST <fsp>
```

and they are appropriate to files which consist only of a string of ASCII characters. This does not include the normal BASIC program files in which the BASIC keywords are replaced by single tokens.

```
*SPOOL <fsp>
```

enables the user to send all that appears on the screen to an ASCII file. This can be the listing of a BASIC program, for instance. The output to the file is terminated by the command

```
*SPOOL
```

without any file name, or by the use of

```
CLOSE#0
```

which closes all open files.

Once a BASIC program has been put into an ASCII file it can be accessed by a word processing system for editing purposes, or it can be merged with other sections of text or with other BASIC programs. If the ASCII file is subsequently required for use as a program again then the *EXEC command can be used.

```
*EXEC <fsp>
```

will cause the contents of the ASCII file specified to be treated just as though it were direct keyboard entry.

The use of the *SPOOL and *EXEC commands is described in detail in Section 9.7.

```
*BUILD <fsp>
```

enables text to be entered into an ASCII file directly from the keyboard.

The text in a file can be displayed by the use of

```
*TYPE <fsp>
```

which will cause the ASCII text version of the file to be sent to the screen. This should only be used with ASCII files as the codes stored in BASIC and

other numeric data files may corrupt parts of the computer memory and (temporarily) disable the computer.

Another format for the display of ASCII files is obtained with

```
*LIST <fsp>
```

which gives a numbered listing of the ASCII file. Each line is numbered sequentially as it is displayed.

The complete contents of any file, whether binary, program or text, can be displayed with

```
*DUMP <fsp>
```

which displays each byte of the file in hexadecimal and also gives the ASCII character where possible.

8.5 TURNKEY SYSTEMS

In some situations it is convenient to start the execution of a program, often the MENU program for a complete disc, by a single keyboard operation. A program or a suite of programs is called a turnkey system if all the user has to do is switch on and load the disc for the software for his problem to be in place and ready to run.

The BBC microcomputer provides an auto-boot facility for which the required initialization action is to press <SHIFT-BREAK> and then release the <BREAK> key first. Once started, the auto-boot facility interacts with the file !BOOT, if it exists, on the disc in drive 0. The form of the interaction depends on the option setting of the disc which may be 0, 1, 2 or 3. The use of the word boot here arises from early computers where in order to start them it was necessary to load a special program which enabled the system to get itself going. This process is analogous to the idea of lifting yourself by your own bootstraps, hence this was called the 'boot' program.

The disc option is set by the *OPT command followed by two numbers separated by spaces. The first first number should be 4 to indicate a change of disc option and the second number should be the required option 0, 1, 2 or 3. Table 8.2 summarizes the options.

Table 8.2 Effects of *OPT 4.

*OPT 4 *option*	*Action*
0	Normal <BREAK>
1	*LOAD !BOOT
2	*RUN !BOOT
3	*EXEC !BOOT

The action following <SHIFT-BREAK> depends upon the previous setting of the disc option. For example

```
*OPT 4 0
```

sets the disc option to 0. The 0 option means that the auto-boot facility is switched off and <SHIFT-BREAK> behaves like <BREAK>.

Disc option 1 will cause the file !BOOT to be *LOADed on operation of the auto-boot facility. If the disc option is set to 2 the !BOOT file will be *RUN. The file !BOOT will need to be a machine code program for this to be used. Probably the most widely used setting of the option is 3, which causes the !BOOT file to be treated as though the *EXEC command had been given for it.

The correct operation of the auto-boot action is conditional on the existence of the appropriate !BOOT file on the disc and on the setting of the appropriate *OPT 4 command. It is perhaps unfortunate that there is not an option to CHAIN"!BOOT" as a BASIC program, since this is very frequently what is really required.

For turnkey systems, the only way to auto-boot a menu or other start-up program is to use disc option 3. To use this option the !BOOT file needs to contain an ASCII version of the immediate mode commands to start the system. An ASCII !BOOT file will normally be created by the use of the *BUILD command.

For example, consider the operation of the command *BUILD !BOOT. As each line is typed, it will be given a line number, and on completion, pressing the ESCAPE key will transfer the file to the current drive. The !BOOT file would be created by the sequence of commands

```
*BUILD !BOOT
```

1. `*TV 0,1`
2. `CHAIN "MENU"`
3. <ESCAPE>

If the disc option is now set by the line

```
*OPT 4 3
```

then on operation of the auto-boot by pressing <SHIFT-BREAK> and releasing <BREAK> first, the *TV command to remove interlace of the video monitor is executed and the program "MENU" will be loaded and run.

In the creation of an EXEC file such as the above !BOOT file it is not necessary to follow the commands by ‖M (the symbols for <CTRL-M> which are equivalent to <RETURN>) as is done with the *KEY commands. The *EXEC command treats each line of the file as though it had been entered from the keyboard and followed by <RETURN>. Thus it would be possible to issue the command

```
*EXEC !BOOT
```

to start a menu driven system with the above !BOOT file and an appropriate MENU program.

Chapter 9

FILE HANDLING

9.1 TYPES OF FILE

The tape and disc filing systems on the BBC microcomputer are not restricted merely to storing BASIC programs. It is also possible to create files to contain data. The data could represent numerical information or text, and if required both types can be stored in a single file. This type of file is frequently called a textfile or data file and since, on some other computer systems, textfile is used to mean a file more like the ASCII file, we shall here refer to such files as data files.

It is worth pointing out straight away that the BBC computer operating system makes no intrinsic distinction between different types of file. They are all stored in the filing system, be it tape, disc or ROM, in exactly the same way.

Thus, *any* file can be loaded into memory (if there is room) by the command

```
*LOAD <filename> (<load address>)
```

If the file happens to be a BASIC program, and the load address defaults to, or is given as, the current setting of PAGE, then it will be loaded and can be run just as if it had been loaded with the BASIC command LOAD "<filename>".

Similarly, you can use the command *SAVE to save a BASIC program, if you know the right addresses to use.

What really creates an effective distinction between different types of file is the way that the information in the file is structured. This structuring will be carried out by the commands used to create the information or to store the information in the file. We can in this sense distinguish at least five separate file types

BASIC program
machine code program
binary file
data file
ASCII file

BASIC programs and machine code programs must have the information structured as the appropriate type of program *before* saving to a file.

Data files and ASCII files have their information structured (differently) by the commands which handle the respective types of file.

Binary files can contain *any* type of information, including the other four file types and also otherwise incomprehensible information such as graphics dumps.

We can deal fairly rapidly with the types of command associated with three of the file types.

BASIC programs

These are normally created by SAVE and loaded by LOAD or loaded and run in a single operation by CHAIN. These three commands are BASIC commands rather than operating system commands, so they are not preceded by a star, and the filename must be in between inverted commas.

Machine code programs

These can only be saved and loaded with the commands *SAVE, *LOAD and *RUN. The last command is equivalent to *LOAD followed by CALL, and plays a role similar to CHAIN for BASIC programs.

Binary files

Any section of memory, such as the area of a graphics display, can be saved and loaded using *SAVE and *LOAD (use of *SAVE and *LOAD are described in detail in Section 10.8.5). Thus to save a graphics picture in Mode 0, 1 or 2, the command would be

```
*SAVE GRDUMP 3000 8000
```

The picture could be loaded back simply by

```
*LOAD GRDUMP
```

The final two filetypes require special commands, and these are dealt with in the following sections.

Exercise 9.1

Enter a very short BASIC program such as

```
10 PRINT "*SAVE test"
20 PRINT "program completed"
```

Save it to disc using the command

```
*SAVE TESTPG 1900 19FF
```

(assuming that you have a disc interface in your computer, so that &1900 is the normal program start position)

Type NEW to clear the program, then load it back again with

```
*LOAD TESTPG
```

LIST and RUN the program.

Exercise 9.2

Switch into Mode 1 and generate a simple graphics display, by commands such as

```
MODE 1
MOVE 1,1
DRAW 1000,1
DRAW 1000,1000
DRAW 1,1000
DRAW 1,1
DRAW 1000,1000
MOVE 1000,1
DRAW 1,1000
```

Use the commands given above to save the file with *SAVE, clear the screen with CLS, and then restore the display with *LOAD. (Note, however, that an interesting effect occurs if the screen display is scrolled between *SAVEing and *LOADing.)

9.2 HANDLING DATA FILES

9.2.1 Files and buffers

In BASIC, a set of commands is provided to facilitate handling of data files. Most of the commands can be used with cassette, single-user disc or level 2 Econet systems, and even the less common filing systems such as Prestel and IEEE.

One important difference between the handling of, say, a BASIC program file and a data file is that in the former case the saving or loading of a file is required to be carried out as far as possible as a continuous operation in the minimum of time, whereas for a data file these operations may extend over long periods. For example, in a data logging application, it may be necessary to record measured values of parameters in an experiment or industrial process at intervals of minutes or hours over a period of days. Similarly, where a file contains a large amount of data to be processed during execution of a BASIC program, it will be necessary to read the data in at irregular intervals as and when required by the program.

In order to minimize the number of disc operations, which relative to RAM operations are very slow, a section of memory is allocated for transfer purposes. This section is known as a buffer and has a capacity of 256 bytes, equivalent to one sector of the disc. All transfers between the computer and the disc must go through this memory. For writing to a file, the data in the buffer is transferred to the disc only when the buffer is full or at the end of file handling when the file is closed. In reading from a file, the buffer is filled and

no subsequent disc operation is needed until all the data in the buffer has been used by the program in INPUT or other statements.

The operation of setting up a buffer associated with a particular file is commonly referred to as *opening* the file. When operations on the file are completed, the buffer must be de-allocated which, not surprisingly, is called *closing* the file.

On the BBC computer, up to five files may be open at any one time; thus up to five buffers may be in use simultaneously.

This is one reason why you should always close a file as soon as transfer of data is finished. A much more important reason is that, when a file is being written to, the last, part-full buffer of information is not sent to the file until it is closed.

While a file is open, information is retrieved from or sent to the file by special versions of the normal input and output commands.

9.2.2 The BASIC commands to open, close and handle data files

The first step in handling a data file is to open the file for reading or writing. This is done by identifying and setting aside the buffer to be associated with a particular file by means of a *channel number*. The channel number is allocated by the operating system, and should be stored in a numeric variable (either integer or real) which will be referred to as the *channel variable*.

The channel number appears always to take the values 17 to 21 for the five files that may be open at any one time, but its actual value is immaterial to the user. The channel variable will always allow you to specify the file you want to work on simply and unequivocally.

OPENOUT

To open a file for writing (that is, to send output to the file), the function OPENOUT is used. OPENOUT returns the channel number to be associated with the file given as the argument to OPENOUT. The channel number should be assigned to a channel variable by a statement of the form

```
<channel variable>=OPENOUT(<filename>)
```

For example, if some data is to be written to a file called DFILE then the line

```
CH%=OPENOUT("DFILE")
```

can be used to get the system to allocate a buffer. (Note that as with most functions, brackets are conventionally included, but are actually unnecessary.)

The channel number to be associated with the file DFILE is allocated to the variable CH%, and CH% should be used with all subsequent file handling operations to inform the system that DFILE is the file to be written to.

A point to note is that with the disc filing system, the command OPENOUT erases any existing version of DFILE, if it exists. If it does not exist, then a file named DFILE, 64 sectors long, is created. The filename given as the argument to OPENOUT may be either a string, such as "DFILE", or a string variable.

Strictly speaking a number could be used for the channel instead of a

variable, provided you were quite certain that you knew in advance what the correct number was for a particular channel, but it would be pointless as well as foolhardy since variables are actually more efficient than numbers within a program. Again it is possible, in principle, to open a file within one program, or even in direct mode, and use it within a subsequent program, but it is most unlikely that you will ever have reason to do this in practice.

PRINT#

The command to write information to a file has the form

```
PRINT#<channel>,variable,variable,...
```

For example

```
PRINT#CH%,A,B%,C$,"DATA"
```

Any variable, expression or value may be sent as output, just as with the normal PRINT command, but in other respects PRINT# differs. Items must only be separated by commas, and some commands such as SPC() and TAB() cannot be used.

CLOSE#

To empty and de-allocate the buffer, when writing is completed, the command required is

```
CLOSE#<channel>
```

For example

```
CLOSE#CH%
```

If instead of the channel variable the value 0 is used for the channel in the CLOSE command, then *all* open files will be closed. This is particularly valuable in an ON ERROR routine, to ensure that all files are closed if a program fails during execution. If you have not included such an error trap, the CLOSE command can also be used in direct mode for the same purpose, either specifying the actual file by its channel variable, or closing all files with a zero. It must be issued before the program is run again, or an error will occur when the computer attempts to reopen the still open file.

Example 9.1

The following procedure would write to a data file the results of a program which are stored in an array A(1) to A(20)

```
500 DEF PROC_writedata(F$)
510 CH%=OPENOUT(F$)
520 FOR I=1 TO 20
530 PRINT#CH%,A(I)
540 NEXT I
550 CLOSE#CH%
560 ENDPROC
```

A simple program to run this subroutine would be

```
10 DIM A(100)
20 FOR I=1 TO 20
30 A(I)=I*I: REM DATA IS FIRST 20 SQUARES
40 NEXT I
50 PROC_writedata("DFILE")
60 END
```

The procedure could equally well handle strings or integers if A(I) were replaced by A$(I) or A%(I).

Exercise 9.3

Write a program to store in a data file a list of the names of cars. Use the INPUT statement to read in the number of names and the names themselves. Run the program for, say, 5 names and use the *CAT command to verify that the data file has been created.

OPENIN

Once a file exists, it can be opened for reading with the OPENIN function, which is used in the same way as OPENOUT, as follows

```
CH%=OPENIN("DFILE")
```

Note that if DFILE does not exist, it will not be created. CH% will be set to zero and all will appear to be well, but when a command tries to use the channel number, it will fail with the error message

```
Channel at line...
```

This is particularly confusing because it is not really that line that is in error, but the line where the OPENIN was issued. It is useful in a situation like this to PRINT the value of CH% in immediate mode. If the value is zero it will confirm that a non-existent file is the source of error.

The fact that CH% is assigned a value of zero if the file does not exist could be used as a safety check, when a new file is to be created, to ensure that a file of the same name does not exist already before issuing an OPENOUT that would destroy an existing file.

INPUT#

Just as with output, the command to read data back from a file parallels the INPUT command

```
INPUT#CH%,A,B%,C$,D$
```

(Obviously, input has to be directly stored into variables, whereas values and expressions can be output to a file. Also, INPUT# is not a function, so it could not be used in an expression such as X=Y+INPUT#.)

Example 9.2

The equivalent to the procedure PROC_writedata, to read the data back, would be

```
500 DEF PROC_readdata(F$)
510 CH%=OPENIN(F$)
520 FOR I=1 TO 20
530 INPUT#CH%,A(I)
540 NEXT I
550 CLOSE#CH%
560 ENDPROC
```

It could be used with a program of the form

```
10 DIM A(100)
20 PRINT "The first 20 squares are"
30 PRINT '"      Number    square"'
40 PROC_readdata("DFILE")
50 FOR I=1 TO 20
60 PRINT I,A(I)
70 NEXT I
80 END
```

End of File

It may well be that you do not know how many items of data there are in a file. There is a function, EOF#, which is designed for this purpose. It has the form

```
EOF#<channel>
```

and returns the value TRUE (−1) if the end of the file has been reached, and FALSE (0) otherwise. This is ideally suited to REPEAT...UNTIL loops, where it can be used as the UNTIL test.

Example 9.3

The procedure and program of Example 9.2 could be modified as follows to read back any number of values

```
 20 PROC_readdata("DFILE")
 25 I=I-1
 30 PRINT "The first ";I;" squares are"
 40 PRINT '"      Number    square"'
 50 FOR J=1 TO I
 60 PRINT J,A(J)
 70 NEXT J
515 I=1
520 REPEAT
530 INPUT#CH%,A(I): I=I+1
540 UNTIL EOF#CH%
```

Exercise 9.4

Write a program to read the names of the cars stored in the data file from Exercise 9.3 and print out the list. Use the EOF# function to determine when the end of the file has been reached.

9.3 FURTHER ASPECTS OF DATA FILES

9.3.1 OPENUP

There are several reasons why you might want to open a file for writing without deleting the old version of the file. There is a third way to open a file, using OPENUP, which opens a file for UPdating (including both reading *and* writing) and does not delete the file first. The function is used in the usual way, for example

```
CH%=OPENUP("DFILE")
```

A serious complication occurs at this point between the two versions of BASIC that have been released by Acorn. These two versions are generally referred to as BASIC I (or just BASIC) and BASIC II respectively. To find out which version of BASIC your computer has, if you do not already know, press <BREAK> and then type

```
REPORT
```

The computer will respond either

```
(C)1981 Acorn    or    (C)1982 Acorn
```

The former message is issued by BASIC I, and the latter by BASIC II, which links the two dates very conveniently to the two versions of BASIC.

The above description of OPENOUT, OPENIN and OPENUP refers to BASIC II. Only OPENOUT is the same for BASIC I.

The other two commands are replaced by a single command, OPENIN, but worse is to follow. OPENIN in BASIC I has the same effect as OPENUP in BASIC II. There is no equivalent to the effect produced by BASIC II's OPENIN. Furthermore, a program containing OPENIN written with BASIC I will list the command as OPENUP on BASIC II. A program with OPENIN written with BASIC II will list *nothing* on BASIC I, and the command will cause an error.

If this seems totally confusing, Table 9.1 may help.

Table 9.1 Summary of commands to open files in BASIC I and BASIC II.

BASIC I	*BASIC II*	*Token*	*Action*
OPENOUT	OPENOUT	&AE	Delete existing file and open file for writing only
N/A	OPENIN	&8E	Open file for reading only
OPENIN	OPENUP	&AD	Open file for reading, writing or updating

(For the technically minded, what happens is that BASIC I and BASIC II use the same token – &AD – for OPENIN and OPENUP respectively. This is very sensible, since the action is the same in both cases, though the BASIC name is different. BASIC II uses a different token – &8E – that BASIC I does not recognize, for OPENIN.)

To make maximum use of the difference between the commands, and to minimize confusion, OPENUP will be used whenever a file is to be opened for both reading and writing. BASIC I users should remember that they must use OPENIN instead.

Programmers writing programs in BASIC II that may be used with either version of BASIC should *always* use OPENUP, even when opening a file for reading only.

9.3.2 Appending to a file: EXT# and PTR#

It is possible, but not very likely, that you may want to change the contents of a simple data file. Random access files, as discussed in Section 9.4, are really needed for this purpose. What you might well want to do, however, is to add further information to the end of a file, or *append* to a file.

Two further file handling commands will enable you to achieve this. The first of these is the function PTR#, which behaves like a pseudo-variable. As you write to, or read from, a data file, the computer keeps track of where you are up to by using a *pointer*, a variable which contains the current position in the file, measured in terms of the number of bytes from the start. This variable is accessible to you as PTR#<channel>, and can be both read and, if you wish, altered.

The second command needed is EXT# (short for EXTent). EXT# <channel> is a function that returns the current size of the file associated with the specified channel.

We can now append to a file, by opening it with OPENUP and setting PTR# to EXT#. (Note that appending is not possible, and the commands EXT# and PTR# will not work, with the cassette filing system.)

Example 9.4

The following program will append the next ten square numbers to the file DFILE, created in Example 9.1.

```
100 CH%=OPENUP("DFILE"): REM USE OPENIN WITH BASIC I
110 PTR#CH%=EXT#CH%
120 FOR I=21 TO 30
130 PRINT#CH%,I*I
140 NEXT I
150 CLOSE#CH%
```

You can test the success of this program by reading back DFILE again with the program from Example 9.3.

9.3.3 Problems with data files on disc

There are two potential problems that can occur when using data files on disc, which is in many other ways much the most satisfactory filing system.

The first is associated with the fact that OPENOUT allocates 64 sectors to a file when it is first created. It may be that a disc has less than 64 sectors free, but ample room for the needs of your particular data file. The program will still not allow you to create the file, failing with a 'Disc full' error.

There is a simple way round this problem, associated with the fact that the operating system does not distinguish between different types of file, as explained in Section 9.1. You can create a 'dummy' file of a different type, and then reopen it with OPENOUT and overwrite it. The 64 sector reservation only applies to a brand new file.

The simplest way to create a dummy file is to use a command such as

```
*SAVE DFILE 0 10
```

that will create a tiny file. It will contain rubbish, but this is immaterial since it is to be overwritten. (You could equally well create a small BASIC program with SAVE.)

The second problem is more serious, and stems from the inefficient way in which the DFS stores its files in strict sequential order on the sectors of a disc. If, as is likely, a new file is saved to a disc following a data file, it will butt up against it on the disc, as shown in Figure 9.1.

The problem now arises that if you wish to append to the data file, there is no room to do so and the program will fail with

```
Can't extend
```

This is an infuriating message when you know that there is plenty of room left on the disc, and there is no foolproof way of avoiding it, except by making the data file the only file on the disc.

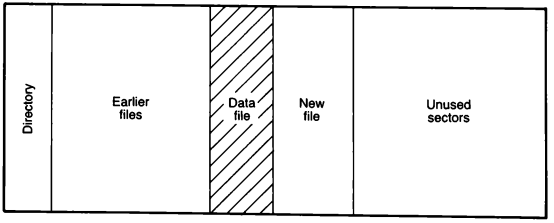


Figure 9.1 The problem with extending a data file.

A reasonably safe, but wasteful, way of ensuring that there is enough room is initially to create a file sufficiently large for all possibilities, by the same *SAVE method a described before. A file of 100 sectors could be created by

```
*SAVE LARGEFL 0 6400     (&64=100 decimal)
```

after which it should only be accessed by OPENUP or OPENIN.

However, you can no longer append to the file by the simple technique described above, since EXT# will point to the end of the 100 sectors, not the end of the genuine data.

Until Acorn change the DFS away from the strict sequential storage method there will be no really satisfactory solution to this 'Can't extend' problem.

9.4 RANDOM ACCESS DATA FILES

Data files can be further subdivided into two subcategories: sequential files and random access files. Up to now we have only discussed sequential or serial access to data files.

As ever on the BBC computer, random access files are simply data files where a very systematic method of organizing the data has been used, so that random access becomes possible. The key to random access is the function PTR# that has already been discussed. Note that random access is only possible with disc-based filing systems.

The simplest way to achieve random access is to organize the data into *records*, which are single or multiple items of data occupying a fixed number of bytes – the record length. Thus if a record length of 40 is chosen, then the Nth record can be found by setting

```
PTR#CH%=40*(N-1)
```

which immediately skips over the previous (N−1) records.

There is a wide range of situations where data can be conveniently subdivided into records – phone lists, names and addresses, journal references, library catalogues, club membership lists, subscription accounts, stock lists, patient records and many similar record-keeping requirements. Quite clearly, anything that can be stored in a card index is ideally suited for a random access textfile.

In most cases, including those listed above, each record will contain more than one item, or *field*. These fields will be accessed sequentially, and the record acts in many respects like a very small sequential data file. In principle, each record could contain a different number of fields, though handling the file would be very difficult unless the records had some systematic structure, such as alternate records with the same numbers of fields.

9.4.1 Creating a random access file

Random access files are handled with the usual data file commands, plus PTR#. However, the record structure must be carefully planned. In particular, the record length must be made sufficiently large for all eventualities.

Consider as an example a simple inventory list, to contain three fields: item name, type and number in stock. The last field is quite simple, because it is numeric, and the internal format used to store numbers is such that integers occupy 5 bytes, and real numbers occupy 6 bytes. (It follows from this that files containing only real or only integer variables could be randomly accessed automatically, since the record length is always 6 or 5 respectively.)

String variables occupy the number of characters of the string plus 2 bytes; thus if the item name and type are considered to need up to 15 and 8 characters respectively, then assuming the stock number to be an integer, the record length required is (15+2)+(8+2)+5=32 bytes. The record does not need to be full for every item; it must contain two strings and one integer but the strings could be shorter than 15 and 8 characters.

It is the standard length of each record that makes random access simple.

Example 9.5

The following program will set up such an inventory list.

```
 10 CH%=OPENOUT("INVENT")
 20 RECNO=0
 30 REPEAT
 40 INPUT "Item name: "ITNM$
 50 INPUT "Type:      "TPE$
 60 INPUT "Quantity:  "QU%
 70 PTR#CH%=RECNO*32
 80 PRINT#CH%,ITNM$,TPE$,QU%
 90 RECNO=RECNO+1
100 UNTIL ITNM$=""
110 CLOSE#CH%
```

The program will loop until <RETURN> is pressed with no other input in response to the item name. (It will also be necessary to press <RETURN> in response to the type and quantity, and this last empty record will be written into the file.)

The key lines are line 70, where PTR# is set to the start of the next record, and line 90 where the record number is incremented.

Exercise 9.5

A danger with the program above is that a name and type longer than the limits set may be entered, which will cause records to overlap. Modify the program to check the length of the fields before writing them to the file.

9.4.2 Reading a random access file

To read records back we need a program which is effectively the mirror image of that in Example 9.5, but it is necessary to know when to stop. As with Example 9.3, the EOF# function can be used for this purpose.

Example 9.6

The following program will read back the inventory produced in Example 9.5

```
 10 CH%=OPENIN("INVENT")
 20 RECNO=0
 30 REPEAT
 40 PTR#CH%=RECNO*32
 50 INPUT#CH%,ITNM$,TPE$,QU%
 60 PRINT "Item name: ";ITNM$
 70 PRINT "Type:      ";TPE$
 80 PRINT "Quantity:  ";QU%
 90 PRINT
100 RECNO=RECNO+1
110 UNTIL EOF#CH%
120 CLOSE#CH%
```

However, the whole point of random access files is to be able to read an individual record without first searching through the whole file and, even more importantly, to be able to modify or edit a record. For instance, with an inventory it must be possible to change the quantity, as items are drawn from stock or reordered. All that is necessary is to set PTR# to the desired record, as explained earlier.

Example 9.7

The following program allows the user to read, or write to, a specified record.

```
 10 PRINT "Do you want to read or write (R/W)? ";
 20 INPUT ""RW$
 30 INPUT "Type the record number that you want: "RECNO
 40 IF RW$="R" THEN PROC_read(RECNO-1): REM SINCE RECORDS
    START AT 0
 50 IF RW$="W" THEN PROC_write(RECNO-1)
 60 END
1000 DEF PROC_read(RECNO)
1010 CH%=OPENIN("INVENT")
1020 PTR#CH%=RECNO*32
1030 INPUT#CH%,ITNM$,TPE$,QU%
1040 PRINT "Item name: ";ITNM$
1050 PRINT "Type:      ";TPE$
1060 PRINT "Quantity:  ";QU%
1070 PRINT
1080 CLOSE#CH%
```

```
1090 ENDPROC
2000 DEF PROC_write(RECNO)
2010 CH%=OPENUP("INVENT"): REM USE OPENIN WITH BASIC I
2020 INPUT "Item name: "ITNM$
2030 INPUT "Type:      "TPE$
2040 INPUT "Quantity:  "QU%
2050 PTR#CH%=RECNO*32
2060 PRINT#CH%,ITNM$,TPE$,QU%
2080 CLOSE#CH%
2090 ENDPROC
```

Exercise 9.6

Modify the above program so that when reading a record, it checks that the end of the file has not been reached.

Exercise 9.7

Modify the program further to allow it to read a record, then change the quantity field for that record.

9.5 SINGLE BYTE ACCESS TO A DATA FILE

9.5.1 The BGET# and BPUT# commands

For most purposes the normal data handling commands INPUT# and PRINT# are quite adequate, but there are circumstances when single byte access to a file is necessary. The commands to read or write a single byte are BGET# and BPUT# respectively. Each operates on the byte pointed to by the current value of PTR#, and PTR# is then incremented to point at the next byte in the file. The format of BGET# and BPUT# are rather different. Bytes are written by

```
BPUT#<channel>,<number/variable>
```

For example

```
BPUT#CH%,10
```

or

```
BPUT#CH%,A%
```

Note that if any value exceeds 255, then only the least significant byte will be written.

BGET# on the other hand returns a value, and must be PRINTed or assigned to a variable

```
PRINT BGET#CH%
```

or

```
BT%=BGET#CH%
```

In the case of both BPUT# and BGET#, the pseudo-variable PTR# is incremented after the operation.

Exercise 9.8

Write a program to display the hex value of each separate byte in the data file from the previous exercises. Run the program and observe the format. An explanation of this format is given below.

9.5.2 Format of data files

Each item of data stored in a data file by the PRINT# command is preceded by a byte which indicates the type of data. A value of &40 indicates integer type and is followed by a further 4 bytes of data, which is the internal representation of the integer number. A value of &FF indicates real type and is followed by 5 bytes of data, the internal representation of a real number. A value of 0 indicates string type and is followed by a byte to indicate the length (L) of the string. A further L bytes follow which are the ASCII values of the characters in reverse order.

In order to position the pointer correctly in a data file containing, say, only integer variables, it is necessary to move the pointer 5 places for each item of data. Similarly for real variables the pointer must be moved 6 places per item. For string variables, however, the situation is more complicated since the lengths in general may not be the same. One solution to this problem is to pad out the strings to the same length by means of spaces at the end of each variable. The following instructions will pad out a string to 10 characters by using the STRING$ function to generate as many spaces as required.

```
10 INPUT "NAME",NAME$
20 L%=LEN(NAME$)
30 IF L%>10 THEN PRINT "Not more than 10 characters": GOTO 10
40 NAME$=NAME$+STRING$(10-L%," ")
```

Exercise 9.9

Write a program to write a list of names of cars to a data file with each string padded out to the same length. Use the program from Exercise 9.8 to verify that it has worked correctly.

Exercise 9.10

Write a program to read from the data file created in the previous exercise any specified item of data. Use an INPUT statement of the form "which car

do you require (1 2 3...)?" and use the PTR# function to locate the required item.

9.6 A GENERAL PROGRAM TO READ OR WRITE A SEQUENTIAL DATA FILE

We are now in a position to write a general purpose program to read or write a sequential data file.

Writing is quite straightforward, using the techniques described in Sections 9.2 and 9.3. For mixed variables, it will be necessary to specify whether each is real, integer or string. Where all are of the same type, this need only be ascertained once.

Reading the file is where a problem arises, because we must cater for any type of data item. This is why this discussion needed to be postponed until this point. We can now use BGET# to interrogate each starting byte of an item to determine which type it is, so that the appropriate type of variable can be used to collect the input. (After the BGET#, the PTR# must be decremented to move it back to the start of the data item.)

Example 9.8

```
 10 ON ERROR GOTO 5000
 20 CLS
 30 PRINT "Do you want to WRITE or READ a datafile?"
 40 INPUT "Please answer R or W: "A$
 50 PRINT
 60 IF A$<>"R" AND A$<>"W" THEN GOTO 40
 70 INPUT "Please type filename: "F$
 80 IF A$="R" THEN GOTO 500
 90 REM
100 REM ******** WRITE DATAFILE ********
110 REM
120 INPUT '"Do you want to append to the file? "YN$
130 YN$=LEFT$(YN$,1): IF YN$<>"Y" AND YN$<>"N" THEN VDU 7:
    GOTO 120
140 CLS
150 PRINT "Do you want to write:"
160 PRINT '"1) All real numbers"
170 PRINT '"2) All integer numbers"
180 PRINT '"3) All strings"
190 PRINT '"4) A mixture of data types"
200 INPUT '"Type the number for your choice: "DTYPE%
210 IF DTYPE%<1 OR DTYPE%>4 THEN VDU 7: GOTO 200
220 MODE 7
230 PRINT CHR$(136);"TYPE <ESC> TO TERMINATE INPUT"
240 PRINT '
250 *FX 229,1
```

```
 260 REM CANCEL ESCAPE
 270 IF YN$="Y" THEN CH%=OPENUP(F$): PTR#CH%=EXT#CH% ELSE
     CH%=OPENOUT(F$)
 275 REM *** USE OPENIN FOR BASIC I ***
 280 REM APPEND IF YN$="Y"
 290 DONE=0
 300 REPEAT
 310 INP$=""
 320 IF DTYPE%=1 OR DTYPE%=2 THEN PROC_number(DTYPE%)
 330 IF DTYPE%=3 THEN PROC_string
 340 IF DTYPE%=4 THEN PROC_mixture
 350 UNTIL DONE
 360 CLOSE#CH%
 370 *FX 229,0
 380 REM RESTORE ESCAPE
 390 CLS
 400 PRINT "Writing completed"
 410 END
 470 REM
 480 REM ******** READ DATAFILE ********
 490 REM
 500 PRINT '"Do you want to pause after each"
 510 INPUT "field? (Y/N): "YN$
 520 YN$=LEFT$(YN$,1)
 530 IF YN$<>"Y" AND YN$<>"N"THEN VDU 7: GOTO 500
 540 CLS
 550 CH%=OPENIN(F$)
 560 REPEAT
 570 DTYPE%=BGET#CH%
 580 PTR#CH%=PTR#CH%-1: REM BACK TO START OF DATA ITEM
 590 IF DTYPE%=0 THEN INPUT#CH%,INP$: PRINT INP$
 600 IF DTYPE%=64 THEN INPUT#CH%,INP%: PRINT INP%
 610 IF DTYPE%=255 THEN INPUT#CH%,INP: PRINT INP
 620 IF YN$="Y" THEN A$=GET$: PRINT
 630 UNTIL EOF#CH%
 640 PRINT '""THAT IS THE END OF THE FILE"
 650 CLOSE#CH%
 660 END
1000 DEF PROC_number(DTYPE%)
1010 PROC_input
1020 IF DONE THEN ENDPROC
1030 NUM=VAL(INP$)
1040 IF DTYPE%<>2 THEN ELSE NUM%=INT(NUM): IF STR$(NUM%)<>INP$
     THEN INP$="": VDU 7: GOTO 1010
1050 IF DTYPE%=1 THEN PRINT#CH%,NUM ELSE PRINT#CH%,NUM%
1060 ENDPROC
1500 DEF PROC_string
1510 PROC_input
```

```
1520 IF DONE THEN ENDPROC
1530 PRINT#CH%,INP$
1540 ENDPROC
2000 DEF PROC_mixture
2010 PRINT "Which data type0 (I-R-S): ";
2020 DTYPE$=GET$: PRINT DTYPE$;
2030 IF DTYPE$=CHR$(27) THEN DONE=1: ENDPROC
2040 REPEAT
2050 UNTIL GET$=CHR$(13)
2060 PRINT
2070 IF DTYPE$="I" THEN PROC_number(2): ENDPROC
2080 IF DTYPE$="R" THEN PROC_number(1): ENDPROC
2090 IF DTYPE$="S" THEN PROC_string: ENDPROC
2100 VDU 7: GOTO 2010
2500 DEF PROC_input
2510 CH$=GET$
2520 IF CH$=CHR$(13) THEN PRINT: ENDPROC
2530 IF CH$=CHR$(127) THEN PROC_cancel: ENDPROC
2540 IF CH$=CHR$(27) THEN PROC_escape: ENDPROC
2550 PRINT CH$;
2560 INP$=INP$+CH$
2570 PROC_input
2580 ENDPROC
3000 DEF PROC_cancel
3010 IF LEN(INP$)=0 THEN VDU 7: PROC_input: ENDPROC
3020 PRINT CH$;: INP$=LEFT$(INP$,LEN(INP$)-1)
3030 PROC_input
3040 ENDPROC
3500 DEF PROC_escape
3510 IF LEN(INP$)>0 THEN VDU 7: PROC_input: ENDPROC
3520 DONE=1: ENDPROC
4970 REM
4980 REM ******** ERROR ROUTINE ********
4990 REM
5000 *FX 229,0
5010 REM RESTORE ESCAPE
5020 CLOSE#0
5030 PRINT "UNEXPECTED ERROR, NUMBER ";ERR;" AT LINE ";ERL
5040 END
```

This program consists of two main parts: the WRITE section starting at line 120, and the READ section starting at line 500. When writing, four data input options are offered: real numbers only; integers only; strings only; or a mixture of data types. Procedures PROC_number and PROC_string handle the actual input of a data item, and for the mixed option, PROC_mixture determines which data type is wanted next and then calls either PROC_number or PROC_string. Typing <ESCAPE> at the start of a data item (but not in the middle) terminates input to the file. There is also an

option to append to an existing file, which opens the file with OPENUP (or OPENIN for BASIC I) instead of OPENOUT, and uses EXT# to move the file pointer to the end of an existing file (line 270).

For reading files, a single option is offered, to pause after each field, so that a long file can be read (an alternative would be to use <CTRL-N>). The first character of each data item is read with BGET# at line 570 to determine the data type. The file pointer is then decremented back to the start of the item at line 580, and the item read into the appropriate type of variable on lines 590, 600 or 610.

9.7 ASCII FILES

A more primitive form of data file is the ASCII file. Whereas items (basically the contents of integer, real or string variables) are stored as distinguishable, structured units in a proper data file, an ASCII file consists simply of a continuous stream of characters stored by means of the ASCII codes of the characters.

Because of the lack of structure, such files cannot be read with INPUT#. However, they can be created and read by certain other system commands, and even by BPUT# and BGET#.

9.7.1 Creating an ASCII file

The simplest way to create an ASCII file is by means of the *BUILD command. This has the form

```
*BUILD <filename>
```

Items are entered one 'line' at a time, where line here means, as with a BASIC program line, the entry up to the point the <RETURN> key is pressed, not a screen line. The computer prompts with a number for each line, but these are not stored in the file. Entry to the file is terminated, and the file closed, by pressing <ESCAPE>.

We shall see shortly that ASCII files are most commonly used as *EXEC files, but in principle *BUILD can be used to produce an ASCII file for any purpose.

An ASCII file can be read quite simply using one of three commands: *TYPE <filename> and *LIST <filename> are almost equivalent. They list back the ASCII file line by line, just as it was created by *BUILD, the only difference being that *LIST adds line numbers, in the same way that *BUILD does, whereas *TYPE does not.

The third command is

```
*DUMP <filename>
```

Its effect is quite different. It produces a hexadecimal dump of the file contents, 8 bytes per line, together with the ASCII representation of each

byte. *DUMP can be used with *any* type of file, not just ASCII files. For instance, it could be used with a data file and would quite conveniently display the format used to store integers, real variables and strings. A specimen *DUMP listing of a data file is shown in Figure 9.2.

```
0000 00 09 53 52 4F 54 53 49 ..SROTSI
0008 53 45 52 00 04 4B 59 2E SER..K9.
0010 33 40 00 00 00 7D 00 00 3@...}..
0018 00 00 00 00 00 00 00 00 ........
0020 00 0A 53 52 4F 54 49 43 ..SROTIC
0028 41 50 41 43 00 05 46 6E APAC..Fn
0030 30 30 32 40 00 00 00 4B 002@...K
0038 00 00 00 00 00 00 00 00 ........
0040 00 0A 53 52 4F 54 49 43 ..SROTIC
0048 41 50 41 43 00 03 46 75 APAC..Fu
0050 31 40 00 00 00 19 00 00 1@......
0058 00 00 00 00 00 00 00 00 ........
0060 00 00 00 00 40 00 00 00 ....@...
0068 00 ** ** ** ** ** ** ** ........
```

Figure 9.2 A specimen *DUMP listing.

9.7.2 *EXEC

One of the main reasons, other than applications such as word processing, for wanting ASCII files is for use with the *EXEC command, which again has the simple form

```
*EXEC <filename>
```

The action of *EXEC is to issue the contents of the specified ASCII file to the computer *exactly as if the contents had been typed at the keyboard.*

There are three common uses for *EXEC.

1 Issuing a series of commands automatically

If you find that you repeatedly issue a particular set of commands, such as programming the red function keys, then these commands can be stored in an ASCII file by *BUILD. Issuing *EXEC for that file will then program the keys automatically.

Exercise 9.11

Use *BUILD to program the function keys to issue a series of commands, such as RUN<RETURN>, LIST<RETURN>, AUTO<RETURN>, RENUMBER<RETURN>, *RENAME, *DELETE. Test the file by *EXECing it. (You should see the commands appear on the screen as they are issued.)

Another example where it can be useful to issue not only commands but program input as well, is to run an interactive program automatically. This

could be useful to set up a file to control a program that takes a very long time to run, or to keep testing a program during debugging.

Exercise 9.12

Set up a file to run the program from Example 4.12 (the class list sort) automatically. You will need to start with

```
CHAIN "EX4.12"
```

then enter the marks one per line.

2 !BOOT files

As was discussed in Chapter 8, it is possible to make a disc into a turnkey system by adding a file named !BOOT to the disc.

The boot option can be set to *EXEC by

```
*OPT 4,3
```

and then, when <SHIFT-BREAK> is pressed, the file !BOOT is automatically *EXECed.

!BOOT can as usual be created by *BUILD. Quite commonly it will only be one or two lines long, say to CHAIN a BASIC program (unfortunately there is no direct boot option to CHAIN a program).

3 Issuing program lines

If a line of the ASCII file starts with a number, then when it is issued by *EXEC it will behave as it would if typed in; that is, it would be stored in memory as a program line. This gives a way of adding procedures or functions to a program, or merging programs, because unlike LOADing a new program section, the existing program remains in memory.

The required program lines could, if you wish, be created by the *BUILD command, but if you already have the program or procedure written, there is an easier way, using *SPOOL.

9.7.3 *SPOOL

The second normal method of creating an ASCII file is by using the *SPOOL command. This has two forms

```
*SPOOL <filename>
```

creates or opens the specified file, and thereafter any output to the screen from a BASIC command is also written into the file.

```
*SPOOL
```

with no filename closes the file or files. (The same effect can also be obtained with CLOSE#0.)

Thus to capture into an ASCII file a program in memory (or part thereof), you would issue the commands

```
*SPOOL TXTPROG
LIST            (or LIST 100,200)
*SPOOL
```

If you subsequently issue *EXEC TXTPROG, all the listed lines will be reissued. Note that the ASCII file contains *everything* that appeared on the screen after *SPOOL TXTPROG, including the commands and BASIC prompts (>), so as well as the BASIC lines you will see the extra lines

```
>LIST     and     >*SPOOL
```

When *EXECed these will generate 'Syntax error' messages just as they would if you typed them (complete with leading >, that is), but this will not affect the program lines that you want.

Merging programs

The best method of merging two programs, or program sections, is by using *SPOOL and *EXEC. The procedure is as follows.

Load the shorter program into memory and capture it into an ASCII file by *SPOOL. Load the longer program into memory, then *EXEC the ASCII file to input the first program back into memory.

Note that if there are any duplicate line numbers the *EXECed lines will overwrite those already in memory. Interleaving lines would be all right in principle, but you are unlikely to want this. One or both programs should be RENUMBERed before merging to avoid this, if necessary.

The most common reason for merging program sections is probably to add to a program favourite procedures or functions, such as error checking or validation procedures. These can be kept as ASCII files and *EXECed into as many programs as you wish.

Editing programs with a word processor

The ability to *SPOOL a program into an ASCII file, and subsequently restore it with *EXEC, gives the possibility of using a word processor to edit BASIC programs. Word processors such as *Wordwise* operate with standard ASCII files, and once a program has been captured into an ASCII file it can be accessed by the word processor. All the powerful word processing facilities such as scrolling back and forth at will, global search and replace, and text movement or duplication can then be used to make program editing much easier. When editing is complete the program can be saved in ASCII form, and then *EXECed to retrieve it as a BASIC program.

Capturing program output in an ASCII file

Although *SPOOL is most widely used for capturing programs, it is not restricted to this. Another possible use is to capture program output.

As an example, consider the way in which short machine code subroutines are sometimes loaded into memory from within a BASIC program, by the ? indirection operator and DATA statements (the indirection operators are described in Section 10.4). The sort of program section required would be

```
1000 FOR J=0 TO 19
1010 READ X
1020 ?(BASEADD+J)=X
1030 NEXT J
1040 DATA 1,2,3,4,5,...19,20
```

The numbers in the DATA statement(s) need to be worked out from the machine code, and if the code is already in memory, this could be done by a program, and the resulting values could be generated as a set of BASIC lines of DATA. *SPOOL can be issued from within a program to create an EXEC file. The following program shows how to do this.

Example 9.9

```
 10 INPUT "Start address of machine code: "SA$
 20 INPUT "Number of bytes: "NB$
 30 SA=EVAL(SA$): REM SA CAN BE IN DECIMAL OR HEX
 40 NB=EVAL(NB$)
 50 INPUT "Line number for start of DATA lines: "LI%
 60 DIM BT%(NB-1)
 70 FOR J=0 TO (NB-1)
 80 BT%(J)=?(SA+J)
 90 NEXT J
100 *SPOOL DATALNS
110 FOR K=0 TO (NB-1)/10
120 PRINT LI%+10*K;" DATA ";
130 J=0
140 REPEAT
150 PRINT ;BT%(J+10*K);
160 IF J<9 AND J+10*K<(NB-1) THEN PRINT ",";
170 J=J+1
180 UNTIL J>9 OR J+10*K=NB
190 PRINT
200 NEXT K
210 *SPOOL
```

Note that in contrast to BASIC commands, the name of the file following *SPOOL cannot be a string variable. This is because *SPOOL, like any

command starting with a star, is an operating system command which is sent directly to the Command Line Interpreter. The CLI cannot interpret BASIC variables, since it operates at a lower level quite independently of whether BASIC is even fitted to the computer. There is a way round this problem, however, as we shall see in Section 10.7.1.

9.7.4 Single byte access to ASCII files

The point has already been stressed that the only difference between different file types on the BBC microcomputer is their internal structure. The general approach to handling data files and ASCII files is very similar. Both operate through buffers, transferring information to or from the filing system a sector at a time. The main difference lies in the special way in which the PRINT# command writes information to the file, so that it can only readily be interpreted by the INPUT# command.

However, this is not true of the single character commands BPUT# and BGET#, which simply transfer a single character to or from the filing system. They can therefore be used to create or read an ASCII file as easily as a data file. This is particularly relevant for reading an ASCII file, since the three normal commands, *LIST, *TYPE and *DUMP all have limitations. *LIST and *TYPE may hang up if they try and read certain control characters, for instance CTRL-B which tries to turn on a printer. *DUMP will certainly list any type of file, but the hexadecimal dump makes it tedious to use with files containing text that you would like to read.

Example 9.10

The following program will write an ASCII type of file, or more accurately any file of characters that can be typed in from the keyboard, including control characters and Mode 7 special effect codes. More importantly it will read them back.

```
10 ON ERROR GOTO 900
20 CLS
30 PRINT "Do you want to WRITE or READ an ASCII"
40 INPUT "file? Please answer W or R: "A$
50 PRINT
60 IF A$<>"R" AND A$<>"W" THEN GOTO 40
70 INPUT "Please type filename: "F$
80 IF A$="R" THEN GOTO 500
90 REM
100 REM ******** WRITE ASCII FILE ********
110 REM
120 INPUT '"Do you want to append to the file? "YN$
130 YN$=LEFT$(YN$,1): IF YN$<>"Y" AND YN$<>"N" THEN VDU 7:
    GOTO 120
140 MODE 7
150 PRINT CHR$(136);"TYPE <ESC> TO TERMINATE INPUT"
160 PRINT '
```

```
170 *FX 229,1
180 REM CANCEL ESCAPE
190 IF YN$="Y" THEN CH%=OPENUP(F$): PTR#CH%=EXT#CH% ELSE
    CH%=OPENOUT(F$): REM USE OPENIN WITH BASIC I
200 REM APPEND IF YN$="Y"
210 REPEAT
220 CH=GET
230 IF CH<>27 AND CH<>127 THEN PRINT CHR$(CH);: BPUT#CH%,CH
240 IF CH<>127 THEN ELSE IF PTR#CH%>0 THEN PRINT CHR$(CH);:
    PTR#CH%=PTR#CH%-1 ELSE VDU 7
250 IF CH=13 THEN PRINT CHR$(10);: REM CR NEEDS LF ALSO
260 UNTIL CH=27
270 CLOSE#CH%
280 *FX 229,0
290 REM RESTORE ESCAPE
300 CLS
310 PRINT "Writing completed"
320 END
500 REM
510 REM ******** READ ASCII FILE ********
520 REM
530 CLS
540 PRINT TAB(0,4);"Option list:"
550 PRINT '''"1) Suppress high bit of each byte"
560 PRINT '"2) Suppress CONTROL codes"
570 PRINT '"3) Send output to printer"
580 PRINT '"4) Pause after each paragraph"
590 PRINT '"5) Start listing"
600 OP%=0
610 REPEAT
620 INPUT '''"Type the number for your choice: "OP
630 IF OP=1 THEN OP%=(OP% OR 1): PRINT '"HIGH BITS WILL BE
    SUPPRESSED"
640 IF OP=2 THEN OP%=(OP% OR 2): PRINT '"CONTROL CODES WILL BE
    SUPPRESSED"
650 IF OP=3 THEN OP%=(OP% OR 4): PRINT '"PRINTER ENABLED"
660 IF OP=4 THEN OP%=(OP% OR 8): PRINT '"PRESS A KEY AFTER
    EACH PARAGRAPH"
670 IF OP<>INT(OP) OR OP>5 THEN VDU 7
680 UNTIL OP=5
690 CLS
700 IF (OP% AND 4)=4 THEN VDU 2
710 CH%=OPENIN(F$)
720 REPEAT
730 CH$=CHR$(BGET#CH%)
740 IF (OP% AND 1)=1 AND ASC(CH$)>128 THEN =CHR$(ASC(CH$)-128)
750 IF (OP% AND 2)=2 AND ASC(CH$)<32 AND CH$<>CHR$(13) THEN
    CH$="."
```

```
760 PRINT CH$;
770 IF CH$=CHR$(13) THEN PRINT CHR$(10);: IF (OP% AND 8) = 8
    THEN A$=GET$
780 UNTIL EOF#CH%
790 VDU 3
800 PRINT ''"THAT IS THE END OF THE FILE"
810 CLOSE#CH%
820 END
900 REM
910 REM ******** ERROR ROUTINE ********
920 REM
930 *FX 229,0
940 REM RESTORE ESCAPE
950 CLOSE#0
960 PRINT "UNEXPECTED ERROR, NUMBER ";ERR;" AT LINE ";ERL:
    VDU 3
970 END
```

When reading a file the program offers a number of options. The first is to suppress the high bit of any byte. Some files may contain characters in a modified ASCII form, having 128 added to the normal code, and this option will strip off the 128.

The second option allows you to suppress any control characters (except the <RETURN> character), printing a full stop instead. The third and fourth options are self-explanatory, allowing you to list to a printer or pause after each paragraph (or more accurately, after each <RETURN>) until any key is pressed. Note that if your printer requires setting up by commands such as *FX 6,0 these will have to be issued before running the program.

One particular use for this program is to read word processor files if you have access to them but do not yourself possess a word processor. Word processor files often contain control codes as well as simple text, and you can use the option to suppress these in order to read the text. The popular word processor *Wordwise* in particular can create files containing CTRL-B codes that will hang the computer unless a printer is connected.

Exercise 9.13

One problem when reading long strings, such as those from word processor files, is that words are broken at the end of each line. Use the type of technique introduced in Example 4.10 to modify Example 9.10 so that when long strings are read, words too long to fit on the end of a line of screen display are wrapped round onto the next line.

Chapter 10

DIRECT MEMORY AND SYSTEM ACCESS

10.1 COMPUTER MEMORY

10.1.1 Hexadecimal notation

The simplest counting system is binary, in which numbers are represented in terms of units, twos, fours, eights and so on, instead of the familiar decimal system where successive digits represent units, tens, hundreds etc. In binary the number nine would be represented as 1001 (1×8+0×4+0×2+1×1) and all numbers are written in binary as a simple string of zeros and ones – two symbols only, which is why it is the simplest, though most long-winded system. This is also why it is used in computers – the two possible values can easily be represented by two alternative states, ON and OFF, or two different voltage levels.

However, binary numbers are very long and inconvenient to work with. The hexadecimal system of numbers is frequently used as a more convenient alternative to binary numbers. This is a number system to base sixteen, or 2^4. The bits (BInary digiTS) are grouped into fours and each group of four is represented by a single hexadecimal digit. For instance, 10010011 becomes 1001 0011 which becomes 93 in hexadecimal (called nine-three because ninety-three implies nine tens and three). A problem occurs with numbers like 10011011 which splits into 9 and 11. We obviously cannot write this as 911 as that represents 100100010001. Conventionally the numbers ten to fifteen are replaced with the letters A to F, in order to represent them by a single 'digit'. Thus the number 10011011 is written, not 911 but 9B in hexadecimal.

Care has to be taken to distinguish between hexadecimal and decimal numbers. 9B is fairly obviously hexadecimal, but the previous example of 93 is not. It is common to indicate that a number is hexadecimal by using a prefix. On the BBC computer the & sign is used for this. Thus our number should be written as &93, which makes it quite clearly hexadecimal. Its decimal equivalent can be calculated by remembering that the 9 represents the 16s column, so it is 16×9+3, or 147. Similarly, &9B is 16×9+11, or 155.

The microprocessor used in the BBC computer is the 6502A, which is an 8-bit microprocessor; that is, it deals with 8 bits of data at a time. This

collection of bits is called a *byte*, and can have values from 0 to 255, or &00 to &FF. Thus the hexadecimal system is a very convenient notation when dealing with information in bytes, since any byte can be written as a two-digit hexadecimal number.

The BBC memory stores information one byte per memory location. In order to be able to locate an individual memory location, it is necessary for that location to be specified by a unique *address*. Hexadecimal numbers are used for addresses, and since 256 memory locations would be quite inadequate, 2 bytes are used to specify a memory address, giving a range of 256^2 or 65536. Memory addresses thus run from &0000 to &FFFF. 65536 is usually referred to as 64K, where $1K=1024=2^{10}$. (The decimal quantities have apparently strange values as they are the equivalents of 'round' binary numbers.)

Computer memory can be one of two types: read only memory (ROM) or read/write memory (RAM – from the less accurate name random access memory). ROM, as the name states, contains data permanently stored on a chip, which can be read but cannot be altered by the computer. The BASIC interpreter and the Machine Operating System are stored in ROM, from &8000 to &FFFF. RAM, on the other hand, is memory which can be read or altered (written to), and this is where BASIC programs, variables and graphics are stored.

10.1.2 Tilde and ampersand

The BBC computer can handle and print numbers in both decimal and hexadecimal form. A tilde (~) in front of a number or expression in a PRINT statement will cause it to be printed as a hexadecimal number (but without the & – since you put the tilde in, you presumably realize that the number is hexadecimal). A number may be given in hexadecimal form by preceding it with an ampersand, wherever an expression can be used. Thus you can PRINT (in decimal) a number given in hexadecimal, or assign it to a variable. You cannot directly INPUT a number in hexadecimal, but you can input it as a string and use EVAL to obtain its value. Try the following example to demonstrate this.

Example 10.1

```
10 REPEAT
20 INPUT "Number: "X$
30 X=EVAL(X$)
40 PRINT "Hexadecimal: ";~X
50 PRINT "Decimal    : ";X
60 UNTIL X=0
```

Run the program first by entering decimal values in the range 1 to 65535 and then in the form &1 to &FFFF. Predict the output for a few values such as 165, &23 and test your expectations using the program. Input a zero to finish.

Note that in Mode 7, the tilde will appear on the screen as a divide sign (remember that normal division is done with the solidus, /).

10.2 ORGANIZATION OF MEMORY IN THE BBC MICROCOMPUTER

It can sometimes be very useful to know how the memory of the BBC computer is organized. Figure 10.1 shows the allocation of the various parts of memory, drawn to scale, and a more detailed diagram of the first section.

In hexadecimal the addressing range of the computer (&0000 to &FFFF) is sometimes regarded as being a series of pages, each &100 (256) locations long, so that the highest two hexadecimal digits give the page number and the lowest two digits give the location within the page.

10.2.1 Memory usage by the system

Important and interesting areas of memory used by the system are as follows

- Zero page (&00 to &FF) A very important area for machine code programs, and used intensively by all parts of the system. &70 to &8F are reserved for the BASIC user, but note that this area may be used by other languages such as Pascal or *Wordwise*.
- Page 1 (&100 to &1FF) Another important area for machine code programmers, containing the 6502 stack.
- Pages 2 and 3 (&200 to &3FF) Used as workspace (places to store values in) by various parts of the system.

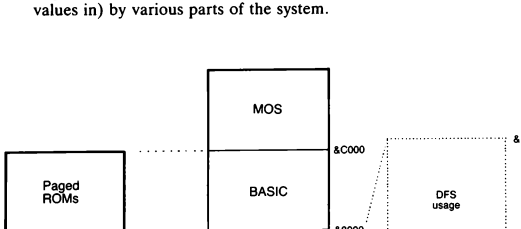


Figure 10.1 Organization of memory in the BBC microcomputer.

- Page 4 (&400 to &4FF) This page is used for two purposes. The first half stores the resident integer variables, and the second half stores the pointers to the linked chains of variables. An individual chain consists of all those variables that start with a particular letter.
- Pages 5 and 6 (&500 to &6FF) Further workspace.
- Page 7 (&700 to &7FF) The input buffer, used to store keyboard input until <RETURN> is pressed.
- Pages 8 to 10 (&800 to &AFF) Mostly used as buffer space for sound, printer, RS423 I/O, cassette I/O and speech. Pages 9 and 10 would be safe to use except when using the RS423 output port, a cassette recorder or speech.
- Page 11 (&B00 to &BFF) This is where the text loaded into the soft keys is stored.
- Page 12 (&C00 to &CFF) This page contains the definitions of the normal user defined characters (224 to 255). (If extra characters are user defined, the start of user space is raised to make room for the extra data.)
- Page 13 (&D00 to &DFF) Used for outside events and by the paged ROMs.
- Address &E00 is the start of user space for the cassette filing system.

10.2.2 Memory usage by the DFS

The disc filing system normally uses up a further 11 pages of memory

- Pages 14 and 15 (&E00 to &FFF) A copy of the current catalogue sectors (track 0, sectors 0 and 1) are stored in these two pages. Note that as long as a disc drive continues to run after a disc access, the DFS assumes that these pages are valid, so if you swap over a disc while the drive is still running, any command issued before the drive stops will use the wrong catalogue information, with potentially disastrous results.
- Pages 16 and 17 (&1000 to &11FF) These two pages are used as DFS workspace.
- Pages 18 to 22 (&1200 to &16FF) These five pages are used as buffers for the five possible files that may be open at any one time (for use with the data file commands or *SPOOL, for example). Note that buffers are *not* used for LOADing and SAVEing BASIC programs or binary files.
- Pages 23 and 24 (&1700 to &18FF) These two pages are used for further workspace.

If a user is desperate for more space for his program, it is possible to recover some of this memory by changing the value of the pseudo-variable PAGE (see Section 10.3). Those pages in excess of the maximum number of files he may want open simultaneously can be used, and if he is not using data or ASCII files at all it is possible to set PAGE as low as &1100, and still be able to load and save BASIC programs with reasonable safety.

The Econet filing system uses just four pages of memory, up to &1200,

unless a DFS is also fitted, in which case the two systems together use memory up to &1B00.

10.2.3 VDU display space

The memory used for screen display depends on the mode, and is summarized in the following list

Modes 0 to 2	&3000 to &7FFF (20K)
Mode 3	&4000 to &7FFF (16K)
Modes 4 and 5	&5800 to &7FFF (10K)
Mode 6	&6000 to &7FFF (8K)
Mode 7	&7C00 to &7FFF (1K)

The top half of memory is allocated to ROMs. &8000 to &BFFF is used by all the paged or 'sideways' ROMs (so called because their memory sits side by side); this includes BASIC and the DFS ROMs. Most of &C000 to &FFFF is used by the Machine Operating System, but &FC00 to &FEFF is allocated for input/output. This memory is used for transferring information to or from the computer through the user port, the printer port and so on.

10.2.4 Program usage

The start of the available memory space depends on the filing system in use, and is referred to as the OSHWM (Operating System High Water Mark). For cassettes it is &E00, for discs &1900, and for Econet it is &1200 or &1B00 if a disc interface is also fitted.

BASIC programs usually start from this point, and build up in memory as required. Variables are stored immediately after the program itself, and BASIC requires some additional workspace, called the BASIC stack, which is normally situated immediately below the memory allocated to the displayed text or graphics. The extent of this memory will depend on the mode being used. Variables can thus build up as far as the bottom of the BASIC stack, which will normally occupy only a small amount of space. If a program, or its variables, tries to expand beyond this point, the error message 'No room' will occur. Trouble can also occur if a large program tries to change mode to one which requires more memory than is available, which will result in the 'Bad Mode' error.

10.3 THE MEMORY POINTERS

There are four important positions in memory for BASIC programs which are accessible from BASIC. The first, and most important, is PAGE, the position of the start of the program. The next is TOP, the position of the end, or top, of the program. The third is LOMEM (LOw MEMory), which is the start of the variable storage area. Finally, HIMEM (HIgh MEMory) gives the last available location in memory for program usage. The BASIC stack builds down in memory from this point.

All of these pointers except TOP are pseudo-variables, and all can be altered as well as read except for TOP.

PAGE normally points to the OSHWM which, as pointed out above, is &E00 for the cassette filing system and &1900 for the disc filing system. However, it is possible to move PAGE either up or down. Note that PAGE (but not the other three pointers) must always be set to a page boundary, that is a value which in hexadecimal has the lowest two digits zero. If you try and set PAGE to any value other than a page boundary, the computer will ignore the lowest two hexadecimal digits. Thus if you tried to set it to &1B80 (7040), it would in fact become &1B00 (6912).

TOP always points to the first free memory location above the program. It is the one pointer which BASIC does not allow to be altered by the user, for fairly obvious reasons. The length of a program can always be calculated by (TOP–PAGE). Note that although you cannot alter TOP by a simple assignment, this restriction can be overridden using the indirection operators described in the next section. If you want to set the value of TOP to NEWTOP, then the following statements will accomplish this

```
?18=NEWTOP MOD 256
?19=NEWTOP DIV 256
```

Great care should be taken in doing this, since setting TOP too low will destroy the end of your program.

LOMEM normally has the same value as TOP, since the sensible place to start the storage of variables is immediately above the program. It can be altered to a different value if there is a reason to do so. You may alter it from within a program if you wish, but only before variables have been assigned, otherwise their values will be lost.

HIMEM points to the next address above the last memory location available to BASIC. It normally points to the bottom of the screen display area of memory, but can be altered, for example to create a safe area for a machine code program. The amount of free space above a program available for variables and the BASIC stack can easily be calculated by (HIMEM–LOMEM).

Exercise 10.1

Write a very short program to give the values of the four memory pointers, of the form

```
10 PRINT "PAGE = ";PAGE
   ...
   ...
```

Also print out the size of program (TOP–PAGE) and amount of free variable space (HIMEM–TOP). (The latter also tells you the amount of space available for program expansion.)

Switch into all the different modes, starting with Mode 7, and notice the dramatic reduction in available space when switching between, say, Mode 7 and Mode 1.

Exercise 10.2

Try altering PAGE to a value other than a page boundary, for example

```
PAGE=&2345
```

then PRINT ~PAGE to see what value has actually been assigned. Press the <BREAK> key to reset the machine, PRINT PAGE again, which will now give the default value, and make a note of what it is for your machine.

Note that whenever you press <BREAK>, PAGE is restored to its default value, so if you alter it deliberately and then load or type in a program, the program will be temporarily lost by a <BREAK>. It can be recovered, however, if you set PAGE once more and then type OLD.

There are two other pointers which it can be useful to read, but which are not accessible as pseudo-variables. These are the pointers to the top of the occupied variable space, and the bottom of the BASIC stack. The pointers are stored in zero page at addresses &02, &03 and &04, &05 respectively. To read them, simply type

```
PRINT "TOP OF VARIABLES: ";?2+256*?3
PRINT "BOTTOM OF STACK: ";?4+256*?5
```

(The use of the indirection operator ? above is explained in Section 10.4.1.) Of more interest than either individual pointer is the size of the gap between them, since this is the amount of free memory available. To calculate this, type

```
PRINT "FREE MEMORY: ";?4-?2+256*(?5-?3)
```

10.3.1 Loading more than one program into memory

One of the uses of PAGE is to allow more than one program to reside in memory simultaneously.

Provided that you set PAGE for a second program above the value of TOP for the first, they will not overlap, and you can move back and forth between them simply by altering PAGE (if the first program needs a lot of space for variable or array storage, you will have to allow for this as well when setting PAGE for the second program, or you could use LOMEM to force the variable space for the first program above the second program).

Exercise 10.3

Enter a short program which prints some message such as 'First program' and test that it runs correctly. Now set PAGE in immediate mode to, say, &3000 and type NEW. Then enter a second program to print 'Second program' and test that. Restore PAGE to the value that you noted down and RUN. The first message should be printed, since that program is still safely stored at that location. Finally, set PAGE to &3000 again and RUN, and the second message should be printed.

A number of short programs can be entered in this way, but note that NEW must be typed after PAGE is changed to a new location for the first time, to

set various pointers needed by the interpreter, or a 'Bad program' error will occur. You must also always be careful that neither program nor variable storage areas overlap, and that you do not set PAGE incorrectly to the middle of a program.

It is also possible to change PAGE within one program, and then to CHAIN another program from within that program. The second program will be loaded at the new setting of PAGE (the new value of PAGE comes into effect when RUN, LIST or some similar command is encountered).

Exercise 10.4

Try the following series of programs

```
10 PAGE=&2000
20 PRINT "First program"
30 PRINT "Now we will run the second program"
40 RUN
```

Now enter two more programs by

```
PAGE=&2000
NEW

10 PRINT "Second program now"
20 PAGE=&3000
30 PRINT "PAGE does not come into effect yet"
40 RUN

PAGE=&3000
NEW

10 PAGE=&1900: REM ASSUMING YOU ARE USING A DISC SYSTEM
20 PRINT "Now back to the beginning"
30 RUN
```

Having entered the three programs, return to the first one and start execution by

```
PAGE=&1900      (or whatever your default value is)
RUN
```

If programs stacked up in memory in this fashion used substantial amounts of variable space, you could use LOMEM to force all the programs to share the memory space above the topmost program, to avoid the possibility of variables from one program corrupting the one above it.

10.3.2 Relocating a BASIC program

It can happen that memory constraints are so tight that a program does not have enough memory when loaded at the normal setting of PAGE. On a

computer fitted with a disc or Econet interface, it is possible to relocate a program after it has been loaded and move it down in memory. To get the maximum amount of space will mean overwriting the disc and/or Econet workspace, but once the program has been loaded this need not matter. The most common reason for needing to do this is if you have a program working on tape, that will not work when transferred to disc. With the DFS the lowest address that you can use without interfering with program transfer is &1100. To retain one buffer for data transfer, you must keep above &1300, and so on. The lowest possible address is &E00, which would render the DFS unusable, but generally it would be sensible only to lower PAGE as far as is necessary for the program to run.

Example 10.2

The following lines, added to the beginning and end of your program, will cause the program to relocate itself automatically when run.

```
    0 N%=&1100: O%=PAGE: REM NEW VALUE FOR PAGE. CHANGE AS
      DESIRED
    1 IF PAGE>N% THEN GOTO 32000
    2 LOMEM=TOP
      ...
      ...
      ...
31990 PAGE=O%: END
32000 A%=TOP-PAGE: REM LENGTH OF PROGRAM
32010 FOR B%=0 TO A% STEP 4: B%!N%=B%!PAGE: NEXT
32020 ?19=?19-(PAGE-N%) DIV 256: REM RESET TOP
32030 PAGE=N%
32040 GOTO 1
```

The main relocation loop must be at the end of the program, because as the program is moved down in memory, the early part will be overwritten. For maximum speed the loop is written on a single line, and uses the resident integer variables, as described in Section 10.6. It also uses the 'pling' indirection operator, which is described in the next section. This is a case where it is also necessary to change the value of TOP.

These program lines could very conveniently be captured into an ASCII file with *SPOOL, and then added to any program requiring relocation with *EXEC.

10.4 MEMORY ACCESS

10.4.1 The indirection operators

Most versions of BASIC provide a means of directly accessing memory by a pair of commands PEEK and POKE for reading and writing respectively. On

the BBC microcomputer, these have been superseded by a set of *indirection operators*, ? (query), ! (pling) and $ (dollar). They act on different numbers of bytes, but otherwise are all identical in operation. Their operation can best be understood by interpreting them as meaning 'the contents of memory location(s)', and they are used for both reading and writing.

To read a byte of memory at location M and print out the result, we can use the command

```
PRINT ?M
```

or if you want the result in hexadecimal

```
PRINT ~?M
```

To understand the meaning of this more clearly, remember the meaning to be associated with ?, and read this command as

```
PRINT the contents of memory location M.
```

Alternatively, the contents of a memory location can be loaded into a variable. For instance, to read the contents of location &FE61 into the variable RD, the command needed is

```
RD=?&FE61
```

Writing to memory is just the reverse of this command; to write a &FF to location &FE61, the command is

```
?&FE61=&FF
```

This time we can understand the purpose of ? as 'the contents of memory location &FE61 (becomes)=&FF'.

Variables can be used for either address or contents. For example

```
?M=WR
```

would write the value of variable WR into location M.

Thus we see that ? serves the purpose of both PEEK and POKE, according to which side of the assignment it is on.

The ! (pling) operator works in exactly the same way, except that it transfers not one but *four* bytes at a time to or from memory.

Note that the value is transferred starting from the *lowest* byte, and that the split between the four bytes will be difficult to work out unless the number is in hexadecimal, when two digits will go into each byte.

Exercise 10.5

Try moving values to and from memory with the ? operator. Memory locations around &2000 would be suitable.

Transfer four bytes into memory locations &2000 to &2003 with the command

```
!&2000=&11223344
```

and then read them back one at a time with ?. Try the same thing with decimal rather than hexadecimal values.

The third indirection operator is the string operator $ (dollar). This transfers a string of up to 255 characters to or from memory. When writing to memory, a string or string variable can be transferred. A <RETURN> character is written at the end of the string to mark its end.

When reading from memory, all the characters up to the next occurrence of a <RETURN> character are read back. Thus a string written by $ can be read back in the same way.

Note that $ is used here in quite a different way from its usual purpose of denoting a string variable. $M means the string contents of memory starting at address M, whereas A$ is a variable which contains a string.

Exercise 10.6

What is the meaning of

1. `$M=A$`
2. `M$=$A`

Exercise 10.7

Write a string into memory by

```
$&2000="123456789"
```

Read it back first by

```
PRINT $&2000
```

and then using the ? operator by

```
FOR J=0 TO 9: PRINT ~?(&2000+J): NEXT J
```

In the latter case you will see the ASCII codes &30 to &39 corresponding to 0 to 9, followed by &0D which is the ASCII code for the <RETURN> character.

10.4.2 Indexed addressing

Both ? and ! have a useful variation which will immediately be recognized by machine code programmers as *indexed addressing*. If either indirection operator is *preceded* by a variable (a number will not do in this context) and

followed immediately by another variable or a number, then the location addressed is the sum of the two quantities. For example

```
ADDR?10
```

means 'the contents of location (ADDR+10)'.

Example 10.3

At first sight this use seems to be confusing and pointless, but consider the following small program

```
10 ADDR=&2000
20 FOR J=0 TO 9
30 PRINT "location ";~ADDR+J;" contains ";~ADDR?J
40 NEXT J
```

In line 30, ADDR?J is exactly equivalent to ?(ADDR+J)

Exercise 10.8

Write a program, using indexed addressing, that prompts for a starting address and a string, and then writes the string into memory as a series of ASCII codes, ending with &0D for the <RETURN> character.

Test the operation of your program by reading the string back with the $ operator.

10.5 A MEMZAP AND DISCZAP PROGRAM

The indirection operators can be used in a MEMZAP program to give easy access to memory, both for reading and modifying. There are a number of possible uses of such a program: the study of how BASIC programs and variable storage are carried out; loading files of any sort with *LOAD and modifying them in memory, after which they can be *SAVEd; or examining the low areas of memory used as buffers. Because the program is very close to what is required for a disc editing program, the same program also provides a DISCZAP facility. Use of this latter application is discussed in Appendix M.

Program ZAP is listed below

```
 10 MODE 7
 20 HIMEM=&7B00
 30 ON ERROR GOTO 30000
 40 DIM BT$(40)
 50 BELL$=CHR$(7)
 60 DIM KEY(12),BYTE$(40)
 70 DATA 136,137,139,138,72,68,65,73,80,82,87,86
 80 REM L-ARROW,R-ARROW,UP-ARROW,DN-ARROW,H,D,A,I,P,R,W,V
 90 FOR J=1 TO 12
100 READ X
```

```
110 KEY(J)=X
120 NEXT J
130 *FX 4,1
140 X=0: Y=0: PG=0: MD%=1
150 DRIVE=0
160 ?&71=0: ?&72=&7B: ?&73=0: ?&74=0: ?&75=3: ?&79=&21
170 BA=PAGE
180 DZAP=0
190 CLS
200 HD$=CHR$(141)+CHR$(130)+"MEMZAP/DISCZAP PROGRAM"
210 PRINT TAB(4,5);HD$
220 PRINT SPC(4);HD$
230 PRINT TAB(10,10);"Do you want:"
240 PRINT TAB(10,13);"1) MEMZAP"
250 PRINT TAB(10,15);"2) DISCZAP"
260 PRINT TAB(10,18);"Type 1 or 2: ";
270 INPUT ""DZAP
280 IF DZAP<1 OR DZAP>2 THEN VDU 7: GOTO 260
290 DZAP=DZAP-1
300 CLS
310 IF DZAP THEN BA=&7B00
320 PRINT TAB(0,21);CHR$(130);"Move cursor:";CHR$(135);
    "[]^v(+SHIFT) ";: IF DZAP=0 THEN PRINT CHR$(130); "New
    Page:";CHR$(135);"P" ELSE PRINT
330 PRINT CHR$(130);"Input:";CHR$(135);"I ";CHR$(131);
    "Input mode:";CHR$(135);"A,H,D ";CHR$(133);"Quit:";
    CHR$(135);"ESC"
340 IF DZAP THEN PRINT CHR$(134);"Read:";CHR$(135);
    "R t,s  ";CHR$(134);"Write:";CHR$(135);"W t,s  ";
    CHR$(132);"Drive:";CHR$(135);"Vn"
350 KEY=5: GOSUB4000
360 IF DZAP THEN TRACK=0: SECTOR=0: GOSUB 6420
370 IF DZAP=0 THEN PROC_printdisplay(BA)
380 PROC_inverse(X,Y)
1000 REM *** HANDLE KEYS ***
1010 PRINT TAB(0,20);"COMMAND:";SPC(32);
1020 PRINT TAB(9,20);
1030 KEY=0
1040 *FX15,1
1050 REPEAT
1060 KEY$=GET$
1070 SHIFT=0
1080 FOR J=1 TO 9+3*DZAP
1090 IF KEY(J)=ASC(KEY$) AND (DZAP=0 OR J<>9) THEN KEY=J:
     PRINT KEY$;" ";
1100 IF INKEY(-1) THEN SHIFT=1
1110 NEXT J
1120 IF KEY=0 THEN VDU 7
```

```
1130 UNTIL KEY
1140 PRINT TAB(0,1);SPC(30): PRINT TAB(11,20);
1150 ON KEY GOSUB 2000,2000,3000,3000,4000,4000,4000,5000,
     6000,6400,6600,6800
1160 GOTO 1000
2000 REM *** L/R ARROW KEYS ***
2010 PROC_normal(X,Y)
2030 IF SHIFT THEN X=7*KEY-7 ELSE X=X+2*KEY-3
2040 IF X>7 THEN X=X-8: KEY=4: GOTO 3030
2050 IF X<0 THEN X=X+8: KEY=3: GOTO 3030
2060 PROC_inverse(X,Y)
2070 RETURN
3000 REM *** DN/UP ARROW KEYS ***
3010 PROC_normal(X,Y)
3020 IF SHIFT THEN GOTO 3500
3030 Y=Y+2*KEY-7
3040 IF Y<0 OR Y>15 THEN Y=Y-16*(2*KEY-7): GOTO 3500
3050 PROC_inverse(X,Y)
3060 RETURN
3500 REM *** CHANGE PAGE ***
3510 PG=PG+2*KEY-7
3520 IF FN_newsector THEN GOTO 6430
3530 PROC_printdisplay(BA+128*PG)
3540 PROC_inverse(X,Y)
3550 RETURN
4000 REM *** SET INPUT MODE ***
4010 MD%=KEY-4
4020 PRINT TAB(33,0);CHR$(131);
4030 IF MD%=1 THEN PRINT"HEX  ";
4040 IF MD%=2 THEN PRINT"DEC  ";
4050 IF MD%=3 THEN PRINT"ASCII";
4060 PRINT CHR$(135);
4070 RETURN
5000 REM *** INPUT DATA ***
5010 INPUT ""IP$
5020 CH=0: BT$=""
5030 FOR J=1 TO LEN(IP$)
5040 CH$=MID$(IP$,J,1)
5050 IF CH$<>" " THEN BT$=BT$+CH$: IF MD%<3 THEN GOTO 5070
5060 IF BT$>"" THEN CH=CH+1: BT$(CH)=BT$: BT$=""
5070 NEXT J
5080 IF BT$>"" THEN CH=CH+1: BT$(CH)=BT$
5090 FOR J=1 TO CH
5100 IF MD%=3 THEN BT=ASC(BT$(J))
5110 IF MD%=2 THEN BT=VAL(BT$(J))
5120 IF MD%=1 THEN BT=EVAL("&"+BT$(J))
5130 PROC_rewrite(BT,BA+128*PG,X,Y)
5140 BT$=CHR$(46): REM DOT
```

```
5150 IF BT MOD 128>31 THEN BT$=CHR$(BT)
5160 PRINT TAB(32+X,Y+3);BT$;
5170 KEY=2: GOSUB 2000
5180 NEXT J
5190 RETURN
6000 REM *** NEW PAGE ***
6010 IF DZAP THEN VDU 7: RETURN
6020 IF MD%>2 THEN PRINT BELL$;CHR$(129);"WRONG MODE";:
     TIME=0: REPEAT: UNTIL TIME=500: RETURN
6030 INPUT ""IP$
6040 PROC_normal(X,Y)
6050 IF MD%=2 THEN BA=VAL(IP$)*256 ELSE BA=EVAL("&"+IP$)*256
6055 IF BA>&FFFF THEN PRINT BELL$;TAB(11,20);: GOTO 6020
6060 PG=0: X=0: Y=0
6070 PROC_printdisplay(BA+128*PG)
6080 PROC_inverse(X,Y)
6090 RETURN
6400 REM *** READ SECTOR ***
6410 INPUT ""TRACK,SECTOR
6420 PROC_normal(X,Y)
6430 IF PG<0 OR PG>1 THEN PG=PG-2*SGN(PG) ELSE PG=0: X=0: Y=0
6440 IF FN_validate(TRACK,SECTOR) THEN PROC_inverse(X,Y):
     RETURN
6450 ?&70=DRIVE: ?&76=&53: ?&77=TRACK: ?&78=SECTOR:
?&79=&21
6460 TRIES=0
6470 REPEAT
6480 TRIES=TRIES+1
6490 A%=&7F: X%=&70: Y%=0
6500 CALL &FFF1
6510 UNTIL ?&7A=0 OR TRIES=10
6520 IF ?&7A>0 THEN PROC_discerror: RETURN
6530 PROC_printsecno
6540 PROC_printdisplay(BA+128*PG)
6550 PROC_inverse(X,Y)
6560 RETURN
6600 REM *** WRITE SECTOR ***
6610 INPUT ""TRACK,SECTOR
6620 PRINT TAB(17,20);BELL$;CHR$(129); "ARE YOU SURE
     (Y/N)?";CHR$(135);
6630 YN$=GET$
6640 IF YN$<>"Y" AND YN$<>"N" THEN VDU 7: GOTO 6630
6650 IF YN$="N" THEN RETURN
6660 PRINT YN$;
6670 IF FN_validate(TRACK,SECTOR) THEN RETURN
6680 ?&70=DRIVE: ?&76=&4B: ?&77=TRACK: ?&78=SECTOR: ?&79=&21
6690 TRIES=0
6700 REPEAT
```

```
6710 TRIES=TRIES+1
6720 A%=&7F: X%=&70: Y%=0
6730 CALL &FFF1
6740 UNTIL ?&7A=0 OR TRIES=10
6750 IF ?&7A>0 THEN PROC_discerror: RETURN
6760 PROC_printsecno
6770 RETURN
6800 REM *** CHANGE DRIVE ***
6810 INPUT ""DRIVE
6820 PRINT TAB(18,0);CHR$(132);"DRIVE ";DRIVE
6830 $&AE0="DRIVE "+STR$(DRIVE)
6840 X%=&E0: Y%=&0A
6850 CALL &FFF7: REM CALL CLI
6860 RETURN
7000 DEF PROC_rewrite(BT,BASEADD,X,Y)
7010 BASEADD?(X+8*Y)=BT
7020 PRINT TAB(3*X+7,Y+3);
7030 IF BT<16 THEN PRINT ;0;
7040 PRINT ;~BT;
7050 ENDPROC
8000 DEF PROC_printdisplay(BASEADD)
8010 LOCAL HORIZ,VERT,BT,BT$
8020 FOR VERT=0 TO 15
8030 PRINT TAB(0,VERT+3);
8040 IF BASEADD<&1000 THEN PRINT ;0;
8045 IF BASEADD<&100 THEN PRINT ;0;: IF BASEADD+8*VERT<&10
     THEN PRINT ;0;
8050 PRINT ;~BASEADD+8*VERT;
8055 IF DZAP THEN PRINT TAB(0,VERT+3);"  ";
8060 FOR HORIZ=0 TO 7
8070 PRINT TAB(3*HORIZ+7,VERT+3);
8080 BT=BASEADD?(8*VERT+HORIZ)
8090 IF BT<16 THEN PRINT ;0;
8100 PRINT ;~BT;
8110 PRINT TAB(HORIZ+32,VERT+3);
8120 BT$=CHR$(46)
8130 IF BT>127 AND BT<255 THEN BT=BT-128
8140 IF (BT>31 AND BT<127) OR BT=255 THEN BT$=CHR$(BT)
8150 PRINT BT$;
8160 NEXT HORIZ
8170 NEXT VERT
8180 ENDPROC
9000 DEF PROC_inverse(X,Y)
9010 PRINT TAB(3*X+6,Y+3);CHR$(130);
9020 PRINT TAB(3*X+9,Y+3);CHR$(135);
9030 ENDPROC
10000 DEF PROC_normal(X,Y)
10010 PRINT TAB(3*X+6,Y+3);CHR$(135);
```

```
10020 ENDPROC
11000 DEF PROC_discerror
11010 PRINT BELL$;TAB(0,1);CHR$(136);CHR$(133); "DISC ERROR
      NO. ";?&7A
11020 ENDPROC
12000 DEF PROC_printsecno
12010 PRINT TAB(0,0);CHR$(134);"TRK ";TRACK;", SECT ";
      SECTOR;"    "
12020 ENDPROC
13000 DEF FN_validate(TRACK,SECTOR)
13010 LOCAL FAIL%
13020 FAIL%=0
13030 IF TRACK<0 OR TRACK>79 OR SECTOR<0 OR SECTOR>9 THEN
      VDU 7: FAIL%=1
13040 =FAIL%
14000 DEF FN_newsector
14010 LOCAL FAIL%
14020 IF DZAP=0 OR (PG>=0 AND PG<2) THEN =FALSE: RETURN ELSE
      FAIL%=1
14030 SECTOR=SECTOR+SGN(PG)
14040 IF SECTOR<0 OR SECTOR>9 THEN TRACK=TRACK+SGN(SECTOR):
      SECTOR=SECTOR-10*SGN(SECTOR)
14050 =FAIL%
30000 *FX 4,0
30010 PRINT TAB(0,23)
30020 END
```

The key lines are 7010, where a modified byte is rewritten to memory, and 8080 where a byte at a time is read from within a double loop over VERT and HORIZ.

The program's output is used as follows: 128 bytes of memory are displayed in the main portion of the screen, in rows of 8 bytes. On the left is the address of the first byte in the row, then the 8 bytes are listed in hexadecimal and finally, where possible, the ASCII character corresponding to each byte is listed. Where there is no printable ASCII character, a dot is shown, and note that the high bit is ignored for this purpose.

This last listing is in many ways the most useful. It enables you to pick out the contents of data files; find your way through BASIC programs; and to determine which parts of memory directly represent meaningful combinations of characters.

Near the bottom is the command line, and below that a brief reminder of the active keys and their uses.

One of the bytes is highlighted in green (on a monochrome screen this will appear as a paler shade), and the highlight can be moved around the screen using the four cursor keys. If the highlight moves off the top or bottom of the screen, a new 'page' of 128 bytes is displayed. The highlight can be moved by a complete page by holding down the SHIFT key while pressing the up or down arrow keys, and to the left or right edge by SHIFT and left or right arrow.

Larger movements can be made by the letter P (for Page), followed by the new page number wanted. For example, to read the memory starting from address &E00, type

```
PE <RETURN>
```

The highlighted byte can be altered by typing I (for Input), followed by the new contents of one or more bytes. Input can be in one of three modes: hexadecimal, decimal or ASCII. The current input mode is displayed at the top right of the screen, and the chosen mode is selected by typing H, D or A. Multiple bytes, starting from the one highlighted, can be input in hexadecimal or decimal by separating the bytes by a single space (not a comma, since the INPUT command is used to collect the string of bytes). In the ASCII mode, the ASCII codes of single characters are written to each byte, and spaces between bytes are not necessary. In the case of hexadecimal and decimal modes, the mode also controls the way in which the P command works (ASCII is clearly inappropriate in this context, and is not allowed). The program is terminated by pressing <ESCAPE>.

If you want to use this program to examine another BASIC program, this can be accomplished quite easily by setting PAGE to a different value before CHAINing ZAP. Any value of PAGE up to about &6800 is possible, since ZAP operates in Mode 7.

Use of the program ZAP for disc editing is described in Appendix M.

Exercise 10.9

Use the MEMZAP facility of ZAP to examine the various areas of memory described in Section 10.2.

10.6 THE WAY BASIC PROGRAMS ARE STORED

With an understanding of how BASIC programs are stored in memory, and the MEMZAP program, you can do a variety of things not otherwise possible. For instance, it is possible in some cases to recover from 'Bad program' situations, recover the existing portion of partly overwritten programs, perform some special tricks and so on. It is not possible in this book to go into great detail about all these features, but we can give the essential background information to enable the adventurous programmer to explore these possibilities.

Program lines are stored as a mixture of ASCII codes and 'tokens'. All BASIC keywords are stored in a special form, as tokens. These are simply single byte codes, distinguishable from ASCII codes because they are greater than &80. MEMZAP will show them in the ASCII display as dots. Line numbers following GOTO statements are stored in a special format, starting with &8D (this format is explained in the book *The BBC Revealed* by Jeremy Ruston, published by Interface). The remainder of a line, except for the

initial line number – formulae, strings, variable names and so on – is stored as ASCII codes (this includes all spaces in the line, including any inadvertently added to the end of a line, as can easily happen when screen editing). Each line ends with &0D, the <RETURN> code.

Each line starts with two items of 'housekeeping' information. First is the line number, stored as a two-byte hexadecimal number, high byte first (in contrast to the way in which addresses are normally stored, which is low byte first). Second comes a single byte containing, in hexadecimal, the length of the line, or more accurately, an offset from the start of one line to the start of the next.

The end of a program is marked by &FF in the high byte position of the line number, which would be the first byte of that line. (This shows up distinctively with the MEMZAP program, as a solid white rectangle in the ASCII listing.) When you type NEW to delete a program, all that happens is that &FF is written over the high byte of the first line number (this will be at address PAGE+1 – programs start with a <RETURN> character &0D at PAGE). Thus to restore a program erased by NEW, all that is necessary is to replace the &FF marker with the proper high byte of the first line number – probably 0 – and then type END in immediate mode to get the BASIC interpreter to sort out other internal pointers. This is, of course, what is done automatically by the OLD command, and it is why there is an apparent bug if you use it with a program starting with a line number greater than 255 – the high byte of the line number has been overwritten and is reset to zero by OLD.

Of course, there is no point in recovering a program erased by NEW with MEMZAP when you can do it more simply by the OLD command; but if you accidentally enter a program line, however trivial, before typing OLD, recovery is no longer possible. With MEMZAP, however, you can delete the end of program marker, sort out the pointer to the first line not overwritten, and thereby recover the remainder of the program.

Exercise 10.10

Type in a short program such as

```
10 SUM=0
20 FOR J=1 TO 10
30 SUM=SUM+J
40 NEXT J
```

Change the value of PAGE, run MEMZAP and examine how the program has been stored in memory. Restore PAGE to normal, delete the program with NEW, switch back to MEMZAP and recover the program by replacing the end of program marker with a zero. Return to normal PAGE once more, type END, and check by LISTing and RUNning that you have recovered the program. You could also see whether you can sort out the problem of recovering the program after typing in a new line after NEW (type in – say – 10 REM).

10.6.1 Protecting a program against LISTing

Another example of the useful jobs that can be done with MEMZAP is protecting a program against LISTing. The essence of the technique is to bury the special character 21 (&15) at the end of a REM, or on its own after a colon. This character will switch off the screen display when listing reaches that point. It cannot be inserted in a REM as CTRL-U, because in direct mode it serves a different purpose, as the command to delete the current line. However, if you put in a dummy character after a REM, you can use MEMZAP to change the character to &15.

If you do this and then try and LIST the program, you will see that the remainder of the program is indeed invisible, but the protection mechanism is less than perfect, because it is quite obvious to any inquisitive user where the program stops listing, and the offending line can be deleted or the REM edited out.

The complete solution is to precede the code &15 with enough delete characters (code &7F) to backspace over the whole line. Again these will have to be entered with MEMZAP over a dummy line such as REM-123456789, with the &15 in last place. To complete the process, you should switch display back on again after listing is complete with code 6 in a final REM.

Regretfully this protection mechanism is far from foolproof, because a user can just LIST that part of your program beyond the REM. A better ploy for real protection is to conceal the fact that you have ever used the technique. Identify some key lines in your program, insert a code &15 before and code 6 after, and renumber the lines so that they are packed together between a pair of lines in the normal sequence of 10s. This type of insertion will be almost impossible to track down (without the user having a MEMZAP program of his own).

10.6.2 Variable storage

There are two factors to be considered regarding the storage of variables: the way in which a single variable is represented, and the way in which the table of variables is organized.

Real and integer variables are stored in the same format as in data files (see Section 9.5.2) and strings are stored as ASCII codes, together with a byte giving the string length.

The storage of integers is quite simple. All integers are four bytes long, giving a range of about ±2147 million, and the bytes are stored in sequence, low byte first, following the normal 6502 convention. The high bit of the highest byte is set for negative numbers.

Real variables, on the other hand, are stored in a much more complex fashion. As with the display of numbers in exponent form or scientific notation, the numbers consist of a mantissa and exponent, and they are stored in five bytes, exponent first.

The mantissa for real numbers is a four byte number, stored high byte first (in contrast to integers where the low byte is stored first). The exponent

is adjusted to be such that the most significant bit of the mantissa (the first bit to the right of the point) is always a one. Thus the 'decimal' point is to the left of the leftmost digit and the number is a binary fraction. The final complication is that because the first digit is always a one it is therefore redundant, and it is used instead to represent the sign of the number, being zero for a positive number and one for a negative number. The remainder of the mantissa simply gives the magnitude of the number; it is not a proper two's complement negative number.

For internal storage the exponent is the power of two stored in a single byte in 'excess 80' form; that is, &80 represents 2^0, &81 represents 2^1 and &7E represents 2^{-2}. Another way of looking at this format is as a signed one-byte number with the sign bit reversed, or with an offset of 2^{80}.

Table 10.1 gives a few examples to help to clarify the format.

Table 10.1 The internal storage format for real numbers.

Decimal number	*Binary form*	*Internal storage*
+1	0.1×2^1	81 00 00 00 00
−1	-0.1×2^1	81 80 00 00 00
+15	0.1111×2^4	84 70 00 00 00
−0.1875	-0.11×2^{-2}	7E C0 00 00 00

Now let us consider how the table of variables is organized. The storage of variables comprises three parts: first is a two byte pointer which is the address of the next variable; next comes the variable name in ASCII form, including the % or $ for integer and string variables and followed by a null byte; and lastly the value of the variable itself as described above. The last variable in the list contains a null byte in place of the high byte of the pointer to the next variable. For example, if with no program present there are two direct mode variables, ALPHA1 and ALPHA2% having values 0.5 and 5, these would appear in memory starting at &1902 (assuming a disc-based system), as

```
0F 19   4C 50 48 41 31    00 80 00 00 00 00
pointer  L  P  H  A  1    null
                          byte

00 00   4C 50 48 41 32 25    00 05 00 00 00
pointer  L  P  H  A  2  %    null
                             byte
```

In the case of strings a complication arises because the string may change in length, possibly becoming longer so that it does not fit into the space initially allocated for it. What happens then is that the original string and its storage

space is discarded, and the new string is added to the end of the variable space. Because of this, two extra items of information are stored immediately after the string name. The first is a two-byte pointer to the location of the actual string, and the second is a single byte containing the number of bytes allocated to the string. The actual length of the string, mentioned earlier, is also stored in a single byte after these other two items, so initially it will be at the start of the string proper, but after reassignment it may be separated. The reason for storing the number of bytes allocated is so that, on reassignment of the string variable, the BASIC interpreter can decide whether the new string will fit into the existing space, or whether that space must be discarded and the string appended to the end of the variable space.

In order to speed up access to a variable, they are not all stored in a single list. Instead, a whole series of linked lists is used, one for each letter of the alphabet, so that all variables starting with the same initial letter form a single chain, and the two-byte pointer at the beginning is the address of the next variable with the same initial. The initial letter therefore becomes redundant, and is omitted. Only subsequent characters of the variable name are stored. The first pointer in the chain is stored on Page 4 at address (&400+2*ASC(<letter>)), following the resident integer variables.

The organization of arrays is even more complicated. When an array is DIMensioned, enough storage space is set aside for the whole array, including obviously the appropriate amount of space for each array element (or in the case of string arrays, the array of pointers). For example a real array dimensioned as DIM A(5,10) would have 66 elements needing 66×5 or 330 bytes. In addition, immediately after the array name (which includes the opening bracket to identify it as an array) the offset to the start of the storage space proper is stored, and then the dimension of each index of the array, two bytes each stored in the usual low/high format. (The offset is thus a double byte having a value of twice the number of dimensions plus 1.)

As an example, a real array declared by DIM ARR(5,10) would appear in memory as

00 00	52 52 28	00	05	06 00	0B 00	81 00 00 00 00...
pointer	R R (	null byte	offset	first dim.	second dim.	first value

Here the offset is 5 (twice the number of dimensions – 2 in this case – plus 1) and the dimensions are 6 and 11 respectively (remember that array indexes start from zero).

In addition to variables, addresses of procedures and functions are also stored in the space above a program, in a further pair of linked lists. The entry for each routine, whether procedure or function, simply contains the pointer to the next item in the list, the routine name (including the initial letter) and the address of the point in the program where the routine is to be found. Just as with variables, the entry in the list is made on the first occasion that the routine is accessed when the program is executed. The start of the chain of

pointers for procedures is at &4F6, and for functions is at &4F8.

Readers wanting to gain a thorough familiarity with variable storage will find it useful to use the MEMZAP program to study some actual examples that can be set up by making assignments in direct mode. (This will entail setting PAGE to a different value – say &6000 – to set up some variables, and switching back to the value of PAGE where ZAP is stored to use the program again.)

From this description of the method of variable storage we can note three points which are relevant to the efficient execution of a BASIC program

1. Use different initial letters as far as possible, since this keeps the chains of variables short and enables the interpreter to find the required variable more quickly.
2. If the string stored in a string variable is increased in length, the initial storage space allocated is insufficient. That space is therefore discarded and the new string added to the end of the variable space. This can eventually waste a great deal of memory in certain circumstances. If strings may grow in this way, they should be 'initialized' first, at their maximum eventual length, by setting them equal to dummy strings. This is particularly important in situations such as sorting, where array elements are likely to be repeatedly reassigned.
3. Procedure and function names are stored along with other variable names. This enables the interpreter to find the appropriate procedure or function very quickly, without searching right through the program line by line, as has to be done for every GOTO or GOSUB. For maximum speed of execution it is therefore advisable to use procedures and functions wherever possible.

10.6.3 The resident integer variables

The integer variables @% (the PRINT format variable) and A% to Z% have special properties and purposes. Instead of being stored in the normal variable space above a program, they are located, four bytes each, from addresses &400 to &46B, as was mentioned earlier. This gives these particular variables two unique features.

First, because their positions in memory are fixed, they can be accessed extremely quickly, and these variables should be used preferentially wherever speed of operation is critical. Second, and much more important, because they are not stored in the usual variable space but in the reserved section of memory below the OSHWM, their values are not cleared by any of the operations, such as loading a new program, that clears all other variables and their values (even <BREAK> does not destroy them). This makes the variables ideal for carrying values from one program to another.

Certain of the resident integer variables also have a third property. Their current values are used to initialize certain 6502 registers whenever machine code programs are executed with the CALL or USR commands. These variables, and their purposes, are

A% – transferred to the Accumulator
C% – transferred to the Carry flag
P% – transferred to the Program counter (this is used with the assembler)
X% – transferred to the X register
Y% – transferred to the Y register

(When the values of the variables are greater than 255, it is the low byte of the value that is transferred. In the case of C%, only the lowest bit is transferred.)

Note that you need not be inhibited from using A% etc. in normal BASIC programs, providing that you are not using CALL or USR.

Exercise 10.11

In direct mode, set variables A%=12, A1%=34 (or any other values that you choose). Press <BREAK>, then try and PRINT the values of A% and A1%.

10.7 HANDLING THE FILING AND OPERATING SYSTEMS FROM WITHIN PROGRAMS

10.7.1 Issuing operating system commands from within programs

In Example 9.9 we saw a case where we would have liked to issue an operating system command with parameters not known in advance. In that particular example, we wanted to issue a *SPOOL command with a filename supplied by the user, but other situations can occur; for instance you might want to issue a *SAVE command with addresses calculated from within the program, or issue a *FX command such as *FX 138 with a character determined by the program.

It has already been explained that all commands starting with a star are not handled by the BASIC interpreter but are passed straight to the operating system where they are handled by the Command Line Interpreter (CLI). This is why the problem occurs, because variables or expressions that we might want to use as parameters to the commands can only be handled by the BASIC interpreter (it is also why no further BASIC commands can be included on the line).

The solution to the problem is to pass your command directly to the CLI, instead of letting the leading star force the BASIC interpreter to do this. There are two ways to send commands straight to the CLI, but the easier one is only available with BASIC II, in the form of a new command, OSCLI. With OSCLI you simply form your OS command without the leading star as a string expression following the OSCLI command.

Under BASIC I the OSCLI command is not available. It is still possible to make a direct call to the CLI but you must first load the desired command directly into memory as a string (again without the leading star), then set the resident integer variables X% and Y% to point to the address of the string, and finally CALL the CLI at address &FFF7.

In both cases the fundamental requirement is thus to form the command into a string, and for BASIC I the string can be loaded into memory using the $ indirection operator, as explained in Section 10.4. The exercise below shows how direct access to the CLI is achieved for both versions of BASIC for the particular problem encountered in Example 9.9.

Exercise 10.12

Make the following modifications to the program from Example 9.9 to allow the user to specify the name of the filename for the spooled output.

```
 100 PROC_spool
 220 END
1000 DEF PROC_spool
1010 LOCAL FILENAME$
1020 PRINT "Type filename for the spooled output:": INPUT
     ""FILENAME$
1030 CMD$="SPOOL "+FILENAME$
1040 OSCLI CMD$
1050 ENDPROC
```

or for BASIC I

```
1040 $&AE0=CMD$
1050 X%=&E0: Y%=&0A
1060 CALL &FFF7: REM CLI CALL ADDRESS
1070 ENDPROC
```

Line 1030 forms the required command as the string variable CMD$ (note the space after SPOOL). For BASIC II all that is then needed is line 1040 to issue the OSCLI command.

For BASIC I line 1040 uses the dollar indirection operator to place the string into memory starting at &AE0, which is a fairly safe area of memory. Line 1050 sets the variables X% and Y% to point at this address (low byte in X%), and line 1060 performs the actual CLI call.

You could write a similar procedure for *SAVE or any other command where the parameters are not known in advance. Note that *SAVE requires addresses to be in hexadecimal, whereas you would normally turn a number into a decimal string with STR$. However, you can get a hexadecimal string form by preceding the brackets after STR$ by a tilde, so you might form the string by a line such as

```
CMD$="SAVE "+F$+" "+STR$~(START)+" +"+STR$~(LGTH)
```

10.7.2 Using commands in the function keys from within a program

There are certain occasions when you may wish to issue commands from within a program which either are not allowed or would be overwritten before

completion. For instance, you might wish to load a new program in two sections using the method of merging described in the *User Guide*, and then run the resultant program. You could not do this in the normal way from within a program, because the LOAD command may not be issued from within a program and because, even if you could get round this problem, one of the program sections might overwrite the old program before it could finish the series of instructions.

The solution is to load the necessary series of instructions into one of the function keys, where they are safe from overwriting, and from where they will be issued as direct mode instructions. You could then have the program tell the user to press the appropriate function key to cause the instructions to be issued, but even this is not necessary. The command *FX 138 allows you to load characters directly into the keyboard buffer, which would have the same effect as if the user pressed the key.

As an example, consider how we would carry out the task mentioned above. The original program could program a function key with the *KEY command, then load that function key into the keyboard buffer, and finally END, at which point the keyboard buffer would be polled and the function key found there and executed.

As a sophistication, you could also disable the screen with VDU 21 before this operation, to suppress the potentially confusing display on the screen of the commands being issued from the function key.

The process of merging requires you to LOAD the first program section, then *LOAD the second section at address (TOP−2) for the first section, and finally sort out the internal pointers with a command such as OLD. It would be simplest to work out the *LOAD address for the second section beforehand by issuing the command

```
PRINT ~(TOP-2)
```

with the first section loaded. The resultant value can then be used with the *LOAD command. Obviously the second section of program must have line numbers after those of the first section.

If we suppose that the two program sections to be loaded are called PART1 and PART2, and the load address for the second is &3456, the following lines in the original program would achieve the desired effect.

```
2000 VDU 21: REM DISABLE SCREEN DISPLAY
2010 *KEY 0 LOAD"PART1" |M *LOAD PART2 3456 |M OLD |M RUN |F|M
2020 *FX 138,0,128
2030 END
```

The parameter 128 to *FX 138 is the code for function key f0. Subsequent keys go up in sequence, to 137 for f9. The |F switches the screen display back on after issuing the commands.

Another occasion where it is necessary to load commands into a function key is to achieve true chaining of programs.

10.7.3 The CHAIN command

There are occasions when it may be desirable, or necessary, to *chain* one program from another. Chaining here means running one program from another, with the variables from the first being retained for use by the second program.

The chief use of this is where a program is too big to fit into memory complete. It must then be broken up into modules, and each module may need to use variables set up by an earlier module. In some instances the modules will form a linear chain, control passing from one to the next until the program is ended. In other cases, where the path of execution through the program may vary, the chain may be haphazard, or even circular if the program forms a giant loop.

The command CHAIN on the BBC computer is actually a misnomer, because it does not provide a true chaining facility. When a new program is CHAINed from a previous one, all variables except the resident integer variables are cleared at the start of the new program, whereas chaining normally implies one program being chained to the end of a previous one, with variable values being preserved during the process.

There are at least three ways in which true chaining can be effected. The least satisfactory is to write all the variables that need to be preserved to a data file before passing execution to the next module, and then reading the values back.

The best method, where it can be achieved, is to use the resident integer variables. These were described in Section 10.6, and an important use of them is in preserving the value of variables when chaining programs. If you also need to preserve strings, these could be stored in a safe area of memory (below PAGE or above HIMEM) with the dollar indirection operator ($), and restored by the next program module. The relevant addresses could be transferred by means of the resident integer variables.

10.7.4 Preserving real variables during chaining

If you need to preserve real variables, or a large number of integer variables or arrays, only the method of storing to a data file on disc would work. There is, however, a third method by which variables can be preserved when CHAINing a new program module, but it is 'unofficial', and great care must be taken since the normal protection mechanisms and checks of the BASIC interpreter must be overridden.

All that happens when you CHAIN a new program is that the computer carries out an automatic CLEAR so that the pointers to the variable chains, stored in the upper half of Page 4 of memory, are destroyed and the pointer to the end of the variable space is reset. Provided that the new program is not larger than the old one, or an existing program is not enlarged by editing, the variables themselves remain in memory undisturbed.

The simplest method of preserving access to these variables is not to CHAIN the new program, but to *LOAD it and then use GOTO to jump into the program without CLEARing variables first. The procedure to adopt is as follows

1. To prevent a longer program module overwriting some of the variables stored at the end of a shorter module, first load the largest of the modules, and PRINT the value of LOMEM. The first module in the chain must start with a command setting LOMEM to this value (or a little larger still might be advisable).
2. Each module must chain in the next by loading one of the function keys with a *LOAD for the next module, followed by a GOTO to the first program line (it would be possible to jump to a later line if you wished, say to skip a DIM statement if the arrays already existed). Note that *LOAD must be used rather than LOAD because the latter, like CHAIN, clears all variables.

Example 10.4

The following pair of programs show a simple example of chaining in operation, with a real variable and a succession of array elements being carried from one program to the next.

Program P1

```
10 LOMEM=10000
20 DIM RV(50),ST$(50)
30 J=0
40 VDU 21: REM DISABLE SCREEN DISPLAY
50 *KEY 0 *LOAD P2 |M GOTO 10 |M
60 *FX 138,0,128
```

Program P2

```
 10 REM THIS LINE must EXIST
 20 VDU 6: REM REACTIVATE SCREEN DISPLAY
 30 J=J+1
 40 IF J>1 THEN PRINT "NEW VARIABLE IS ";XYZ
 50 IF J>50 THEN END
 60 INPUT "TYPE A NUMBER",RV(J)
 70 INPUT "TYPE A STRING",ST$(J)
 80 FOR I=1 TO J
 90 PRINT ST$(I),RV(I)
100 NEXT I
110 PRINT: INPUT "NEW VARIABLE",XYZ
120 VDU 21
130 *KEY 0 *LOAD P2 |M GOTO 10 |M
140 *FX 138,0,128
```

10.8 USING MACHINE CODE PROGRAMS

It is not the purpose of this book to teach machine code programming. However, it can be useful for the non-machine code programmer to have

some understanding of what machine code is, and how he can use it without needing to understand it in any detail. For instance, program listings that you wish to copy may include a section of machine code. Nevertheless, some readers may not be interested in machine code at all, and this section may safely be skipped if you wish, apart perhaps for Section 10.8.5.

Machine code is a very primitive language, but it is understood directly by the 6502 microprocessor, whereas BASIC has to be decoded into machine code by the BASIC interpreter which is resident in ROM. This makes BASIC extremely slow compared with machine code for simple operations – typically machine code may operate 100 times faster than BASIC. For some applications only machine code can give the speed necessary. Arcade games are an obvious example, though by no means the only one. (An example of a more serious application where machine code is used is in routines to produce copies of graphics screen displays on a printer, of which several have been published.)

One other instance where machine code has to be used, is to carry out operations which simply cannot be done in BASIC because of the limitations of the BASIC language or interpreter.

True machine code is a series of one byte numbers that are the operating instructions for the 6502 microprocessor. Many of these are followed by one- or two-byte addresses giving the location where data is to be fetched from, or written to. Since these 'op codes' are exceedingly difficult to remember, programmers invariably use mnemonic code, with three letter mnemonics for the op codes. An assembler is used to translate these, in essence by looking up the corresponding numbers from a table. (An assembler also has a number of sophistications to make life a little easier for the programmer, in particular by the provision of labels.)

The operations that are possible are very simple, mainly reading from and writing to memory, adding or subtracting bytes, unconditional jumps or conditional branches, and a variety of operations on individual bits of a byte.

A machine code program is executed by specifying the memory address of the first instruction. This can be done from BASIC, by the command CALL followed by the starting address. The operation codes (with their accompanying data addresses where appropriate) are then executed in order, unless a jump or branch instruction is encountered.

10.8.1 The BBC assembler

One of the many unusual features of the BBC microcomputer is the built-in assembler that works directly from BASIC. In particular, the assembler can handle BASIC variables at any point in the assembly code, substituting into the machine code the value of the variables at the time of assembly.

The assembler is invoked, either in direct mode or more commonly from a program, by an opening square bracket – [(note that in Mode 7, square brackets are printed as left and right arrows). In direct mode, each assembler line must start with a square bracket, but in a program assembly continues until a closing square bracket is encountered (this must be at the start of a line). Within the square brackets, the usual mnemonic forms are used,

together with labels which are denoted by a preceding full stop. Both labels *and* BASIC variables can also be used for addresses.

Very few of the usual pseudo-op codes are provided, because there are other ways of achieving the same operations.

The resident integer variable P% is used to set the program counter from BASIC before entering the assembler, thus defining the origin for the assembly. Most assemblers do this with an ORIGIN pseudo-operation. Similarly, in place of a DATA or DFB pseudo-op, it is possible at any time to leave the assembler temporarily and enter data into memory with the indirection operators. P% can again be used to give the position in memory, provided that it is incremented for every byte entered. Labels can be defined in the same way as ordinary BASIC variables, removing the need for an EQU or = pseudo-operation. (In BASIC II only, there is a set of pseudo-ops EQUB, EQUW, EQUD, EQUS to enter respectively, 1, 2 or 4 bytes of data or a string into memory. These are equivalent to the usual DFB rather than EQU.)

Comments can be included in a statement at the end, if they are preceded by a backslash.

Example 10.5

Enter the following short program

```
10 start%=&C80
20 P%=start%
30 [
40 OPT 1
50 LDA #5
60 ADC #7
70 RTS
80 ]
```

Line 20 sets the start of the assembly to &C80, which is a safe position unless you have a large set of user defined characters. P% could have been used in place of start% in line 10, eliminating line 20, but this form has been used in order to be consistent with a later version of the program. RUN the program and look at the display. OPT is not a true mnemonic but a 'directive' or pseudo-operation telling the assembler what listing OPTion to use. OPT 1 lists everything as the program is assembled for the first time. Working from left to right the listing shows the address, the byte(s) forming the instruction, the mnemonic notation for that instruction and lastly any comments. Instructions need from one to three bytes. It is beyond the scope of this book to describe in detail the instructions executed by the processor – those used in this program are

LoaD Accumulator with the number 5
ADd with Carry the number 7
ReTurn from Subroutine

Note the use of P% to determine the position in memory at which the machine code will be assembled. As was pointed out in the previous section, certain of the resident integer variables have special applications for assembler programs. A%, C%, X% and Y% are copied to the appropriate registers when the machine code routine is called or used. P% is used during the assembly process to set the program counter to the desired position. Note that at the end of the routine the registers are not returned to the resident integer variables, but they are accessible through the USR function, as explained below.

10.8.2 CALL and USR()

There are two ways of running user-defined machine code routines – CALL <address> and USR(<address>), where <address> may be either a number such as &1234 or a variable. CALL updates the 6502 from the relevant resident variables and then jumps to the user routine. The RTS instruction returns to the BASIC system. The changed values of the registers are lost. CALL is the machine code equivalent to PROC which simply does something without returning values to the program.

USR calls up a machine code subroutine in a similar way to CALL, but it is a function rather than a command and it also returns the values of the processor status register, accumulator, and X and Y index registers, as a single 32-bit number.

Exercise 10.13

Run the machine code program created by Example 10.5 by typing

```
PRINT ~USR(start%)
```

(Note that Example 10.5 did not *run* the machine code program. It simply created it.)

You should see an eight character (four-byte) number. The left-hand byte represents the status register which stores 'flags' such as carry, zero and overflow; the next two bytes are the X and Y index registers and the last (right-hand) byte represents the accumulator. This should have the value 0C or 0D depending on whether the carry bit was set (ADC is ADd with Carry). Type C%=1 first then PRINT ~USR(start%) again. The accumulator should definitely be 0D this time. Set X% and Y% to different numbers in immediate mode and print USR again and notice which byte represents which index register. To prove that A% is left unchanged set A% (the accumulator variable) to an arbitrary value and print A% after using USR again.

Exercise 10.14

It is inconvenient to use numbers in CALL and USR statements and it is not necessary when labels have been included in the mnemonic listing. Insert

```
35 .test
```

in the program from Example 10.5 and RUN it again, then try PRINT ~USR(test). The full stop in front of 'test' tells the assembler that a label rather than a mnemonic follows – the dot is not part of the label name. Hence when the label is used as an operand it is written just as 'test', with no full stop in front.

Exercise 10.15

Type NEW and enter the following

```
 10 start%=&C80
 20 P%=start%
 30 [ OPT 1
 40 .test
 50 LDA #7
 60 .loop
 70 ADC #5
 80 JMP loop
 90 RTS
100 ]
110 END
```

RUN the program and look at the code generated for the JMP (JuMP) instruction – 'loop' has been replaced by the actual address of the ADC instruction. (*Do not* try and CALL the machine code program – it is a closed loop and will run until interrupted.)

Interchange lines 60 and 80 so that the jump instruction points 'forward', and run the program to assemble again. Change OPT 1 to OPT 3 and note the error message. When the assembler encounters JMP loop now it does not know what to do as 'loop' has not yet been defined. We must force the assembler to work through (pass) the code twice – once to discover all the labels and then again to assemble properly.

We can make two passes of the assembler by using a FOR...NEXT loop around the code. We are not interested in error messages on the first pass as we will expect problems with labels so we need to use different OPTions for each pass, which is achieved by using a variable as the parameter for OPT. The OPTion number is actually a two bit number and so has the value 0 to 3. The least significant bit enables (1) or disables (0) listing and the most significant bit should be cleared or set according to whether it is the first or second pass to suppress some of the error messages. We will use OPT 0 (no listing) on the first pass and OPT 3 (listing and error messages) on the second pass.

Exercise 10.16

Modify the program from Exercise 10.15 by adding extra lines 15 and 105. The resultant program will be as follows

```
 10 start%=&C80
 15 FOR Z%=0 TO 3 STEP 3
 20 P%=start%
 30 [ OPT Z%
 40 .test
 50 LDA #7
 60 JMP loop
 70 ADC #5
 80 .loop
 90 RTS
100 ]
105 NEXT Z%
110 END
```

Run the program and observe that the listing (which comes from the second pass) now has the forward address for the JMP worked out correctly.

Note that there is nothing special about Z%; any variable would do. Again do not attempt to execute the resultant machine code program. It is a meaningless program simply designed to explore the problems of forward jumps.

Exercise 10.17

Why has P%=start% been included *inside* the FOR...NEXT loop? Test your answer by moving it to line 12 after modifying line 15 to select OPTions 1 and 3 to list on both passes.

10.8.3 Allocating space for a machine code program

For machine code programs needing larger amounts of space than can be safely found in areas such as &C80, it is possible to set aside memory in the variable storage area for the machine code.

```
DIM start% 300
```

would allocate 300 bytes of memory for a routine, the start of which is stored in the variable start%. In the above programs, line 10 could be replaced by

```
10 DIM start% 20
```

Note that this use of DIM is at first sight quite different from the usual form, where a variable name is followed by a value in brackets. In fact the use is not so illogical: when an array is defined, enough memory is set aside for all the elements of the array, so in both cases, DIM has the effect of reserving memory in the variable storage area for later use.

10.8.4 System access

It would be difficult, from within machine code, to carry out such tasks as printing a character to the screen, and in cases such as this it is possible to

access the relevant BASIC interpreter routine from within the machine code program. Particularly easy routines are the *FX commands and the VDU driver routine. *FX commands are invoked by JSR OSBYTE (JSR means Jump to SubRoutine). OSBYTE stands for Operating System BYTE and is at address &FFF4. (You cannot use the word OSBYTE directly in assembly unless you have defined it earlier in the BASIC part of the program using a LET command.)

The particular *FX is selected by the value in the accumulator, X register and Y register. The accumulator specifies which *FX call is to be made and X and Y supply the remaining two values sometimes required. X and Y should be set to zero when no other values are needed so

```
LDA #0
LDX #0
LDY #0
JSR &FFF4
```

correctly assembled and CALLed is equivalent to *FX 0 (print the version number).

Example 10.6

The following program will perform this call.

```
10 DIM P% 10
20 [OPT 1
30 .fx LDA #0
40 TAX
50 TAY
60 JSR &FFF4
70 ]
80 CALL fx
```

(TAX and TAY mean: Transfer (copy) Accumulator to X or Y.) RUNning the above program should print the operating system number.

VDU calls are carried out by JSR OSWRCH, which is at address &FFEE. Where more than one parameter is needed (as is usual with VDU statements) the JSR is simply used repeatedly after loading the accumulator with successive values. (BASIC programs also carry out writing operations by calling OSWRCH.)

Other OS calls that you may wish to use are: OSRDCH (address &FFE0) which reads a character from the currently selected input stream; OSNEWL (address &FFE7) which writes a newline; and OSCLI (address &FFF7) which has been explained in Section 10.7.1. Other more specialized OS calls are OSFIND (&FFCE), OSGBPB (&FFD1), OSBPUT (&FFD4), OSBGET (&FFD7), OSARGS (&FFDA), OSFILE (&FFDD) and OSASCI (&FFE3). These calls are all documented in the BBC *User Guide*.

One simple use of machine code is in fact to access *FX calls which

return parameters. For instance, OSBYTE call &87 (equivalent to *FX 135) returns the character at the text cursor position in the X register, and also the current mode number in the Y register. Thus if you want to ascertain the current mode, for instance to save having to request the mode from the program in a procedure such as PROC_inverse from Chapter 11, the following section of program could be used

```
100 A%=135
110 MD=USR(&FFF4)
120 MD=(MD AND &FFFFFF) DIV &10000
```

Note that it is not even necessary to invoke the assembler in order to make an OSBYTE call. The accumulator is set via A% (the X and Y registers could similarly be set by X% and Y% where necessary) and the result (from the Y register in this case) is read from the USR function.

10.8.5 Saving and loading machine code programs

There are two ways of storing machine code programs. If it is satisfactory to assemble the program every time, it can be stored in assembly form as part of a BASIC program (once development is over, the OPT parameter can be changed to 0 or 2 to suppress the assembly listing).

However, if for some reason this is not desired, the raw machine code can be saved direct from memory with the *SAVE command. This is still possible even if it is to be combined with a BASIC program. BASIC and machine code programs can be stored together by *SAVE, and loaded back with the BASIC command LOAD or CHAIN.

*SAVE simply saves a section of memory to the current filing system. It therefore requires three parameters: the filename, and two parameters which specify the section of memory. The section of memory may be specified in one of two forms. In either case the first parameter given must be the start address of the section with the second being the end address or (if preceded by a + sign) the length of the section. In all cases the parameters must be in hexadecimal. For example

```
*SAVE MCPROG1 C80 D00
```

would save the section of memory from &C80 to &CFF. Note that the last address specified is not saved.

```
*SAVE BASPROG 1900 +400
```

would save the section of memory from &1900 to &1CFF.

There are two further optional parameters to *SAVE. The first of these is the *execution address*. If the section of memory being saved is a machine code program, and it is subsequently executed with *RUN, this address is where execution will start. It is only needed when the execution address is other than the start of the section of memory. Thus if a program is saved with

```
*SAVE MCPROG2 C80 D00 CA0
```

then *RUN MCPROG2 would be equivalent to

```
*LOAD MCPROG2
CALL &CA0
```

The second extra parameter, which will only rarely be necessary, is the reload parameter. Normally the section of memory would be loaded back at the same address as it was saved from, unless otherwise specified by the *LOAD command. But the default reload address can be specified to be elsewhere by the fifth *SAVE parameter.

All of these parameters (except the starting address, if a different reload address is given) can subsequently be determined by the *INFO command once the file has been saved. On the DFS, *INFO MCPROG2 would give the response

```
$.MCPROG2      000C80 000CA0 000080 nnn
```

where nnn is the sector on the disc where storage of MCPROG2 starts. The extra two digits for the addresses are for future expansion of the system. (If the file were locked, this would also be shown in the *INFO listing.)

The file can be reloaded by the *LOAD command, which has only two parameters – the filename, and an optional reload address (in hexadecimal) if you do not want to load the file at the default address. (If you want to *LOAD a data or ASCII file you should always specify the reload address, since the default is &0000 which is unacceptable.)

If the file is a machine code program, then an alternative to *LOAD is to load and run it in one operation, from the execution address specified in the *SAVE command as explained above. This is the machine code equivalent of the BASIC command CHAIN. The command *RUN has the ultimate in abbreviations, to a simple star. Thus instead of *RUN MCPROG2, we could simply use

```
*MCPROG2
```

giving the file all the appearances of an extra OS command. In this form, the file can also be accessed in the library directory (as set by the *LIB command) if it does not exist in the current directory.

Chapter 11

PROGRAMMING TECHNIQUES

11.1 'USER-FRIENDLY' PROGRAMMING

In designing a program it is important to ensure that it will be easy to use effectively – 'user-friendly' is the word used to cover all such aspects. In particular it is important to validate any user input and to use attractive display.

Input

There are numerous of ways in which the programmer can make responses easier for the user. For instance, if all responses are by means of a single key, then it is tedious to have to press <RETURN> after every key-press. Similarly, to move objects around the screen, it is nice to use the four arrow keys, rather than a collection of meaningless letter keys. This means redefining the keys. The various methods of input are described in Section 11.2.

Consistency

The most important point here is that the program should be consistent in the way it expects the user to respond. For instance, if GETs and INPUTs are mixed, the user will not know whether he is expected to press the <RETURN> key, and sometimes having to reply YES and other times Y to a yes/no question is another cause of great frustration, even to a fairly experienced user.

Validation

Whenever the user is asked to make a response to the program, it is important to check both for typing errors, and that the data returned is sensible in the context of the program. This is discussed fully in Section 11.3.

Display

There is a great deal that can be done in this context, which will be covered in detail in Section 11.4. However, at its simplest it includes such techniques as

putting a short message in the centre of the screen rather than in the top or bottom left hand corner. The reaction testing program of Example 11.1 illustrates this.

Information

Within reason, it is desirable that a program should be operable with a minimum of reference to supporting literature, whether this be course notes for a teaching package or an operating manual. This means that the program itself should give as much information as possible. In particular, when asking the user to make a choice it is helpful to explain the significance of those choices. The result may well be a program consisting more of PRINT statements than anything else, but this is not uncommon and should not be a deterrent.

Menu

Whenever a user has to make a choice from a series of options, this should be presented in the form of a *menu*. The word menu is used in analogy to a restaurant menu (particularly Chinese restaurants) where the list of dishes is presented for the customer's choice, with a number against each so that a dish can be selected simply by quoting the number.

In the same way a computer program menu allows the user to select his choice simply by pressing the appropriate key. The key-press will often be detected by means of a GET, so if there are more than 9 choices, it is better to use a letter. This can be converted to a number and then used in an ON...GOTO statement. Another possibility is to use initial letters for selection, even if these are not in sequence. For instance, suppose your menu has four choices with initial letters L, S, R and E (ASCII codes 76, 83, 82 and 69) and the code of the key pressed is stored in variable X. By using relational expressions a variable A could be assigned as

```
A=-((X=76)+(X=83)*2+(X=82)*3+(X=69)*4)
```

and A would then be suitable for use in a statement of the form ON A GOTO...

The methods of using menu programs are described in Section 11.7.

11.2 INPUT FACILITIES OF BBC BASIC

One of the great advantages of a microcomputer is that it is ideally suited to interactive use, since one user has sole control of the machine. This interaction may take many forms: perhaps the most exciting is the computer game, but there are many more serious forms, ranging from simply deciding whether to run a program again to positioning a point on a graphical display (which might be to draw a modified character, or to label a feature on a graph, for instance).

The first essential, whenever a user is expected to interact with a

program, is to display a message at the appropriate time so that he knows when and how to interact. The INPUT command is particularly convenient for this purpose, since it allows a message to be printed as part of the INPUT command.

There are, however, a number of other useful ways in which to allow the user to interact, as detailed below.

11.2.1 Joysticks and buttons

For completeness we will mention here that the analogue inputs of the BBC microcomputer make it simple to use joysticks and buttons or similar devices in conjunction with BASIC programs as well as with games. The function ADVAL(N) will read the Nth analogue input channel (N=1 to 4), while ADVAL(0) will indicate whether one or both games buttons are being pressed. For example, X=ADVAL(1) returns into X the value corresponding to the current reading on channel 1 of the analogue input.

Further details of the ADVAL function and use of the analogue inputs are outside the scope of this book.

11.2.2 The GET and GET$ functions

The standard method of receiving INPUT from the user is via the INPUT command. For input of long numbers or strings this is ideal, because it provides editing facilities, in the form of the DELETE key, to correct mistakes during input. For input of a single number or character, however, INPUT is very tedious, as pointed out above.

There is an alternative way of collecting input, via the GET and GET$ functions. These differ from INPUT in that they will accept only a single number or character. They give no new line or prompt and there is no need to press <RETURN>. (Neither do they display the character typed, but you can always PRINT this if required.) Both functions will accept any key, but GET returns a number, the ASCII code of the key, while GET$ returns the single-character string represented by the key. The commands are issued as

```
A=GET or A$=GET$
```

(Note the difference from some other microcomputers, where GET is a command rather than a function.)

It is possible to use GET$ for multiple character input. Lines 220 to 260 of Example 4.10 use GET$ to input a character string, in the form of an array, which might be much longer than the 255 character maximum of INPUT.

Exercise 11.1

Modify your program from Exercise 4.14(1) earlier to use GET$ instead of INPUT.

It can on occasion be useful, as pointed out above, to be able to use the cursor editing keys, particularly the arrow keys, for normal input. This can be achieved with the command

```
*FX 4,1
```

After this command has been issued, the arrow and COPY keys can no longer be used for screen editing. Instead, when pressed they return normal ASCII codes, as follows

```
COPY    135
←       136
→       137
↓       138
↑       139
```

These keys can be picked up in the usual way with GET$, in a statement such as

```
IF GET$=CHR$(135) THEN...
```

*FX 4,0 restores the keys to their normal function, and note that this must be issued before the end of a program, as the keys will remain disabled after the program has ended. (Note that *FX 4,2 has a second action – it enables the five keys to act as further function keys, with numbers 11 to 15.)

In a similar fashion to *FX 4,1

```
*FX 229,1
```

disables the normal action of the ESCAPE key, and it returns its standard ASCII value of 27. *FX 229,0 restores the normal action.

Exercise 11.2

Write a program to move a character around the screen by means of the four arrow keys, and end when COPY is pressed.

11.2.3 The INKEY function

A feature of both INPUT and GET which is often necessary, but sometimes inconvenient, is that the computer stops and waits for the input. There are times when it is useful, or indeed essential, for the computer to continue the program meantime. (Games are again a very obvious example, where say a gun is occasionally to be fired at a moving target.)

The function INKEY (and the string equivalent, INKEY$) enables the programmer to check for input while a program is running. Whenever a key is pressed it is stored in the keyboard input buffer. The next INKEY function can pick the key value from the buffer when program execution reaches that point.

There are really three separate modes of operation of INKEY. INKEY(N), where N is a positive number, will (if necessary) cause a program to pause for a period of up to N centiseconds to see whether a key is pressed. If a

key is pressed during that time, or prior to the INKEY call, INKEY returns the ASCII code of the first item in the keyboard buffer. After N centiseconds the computer gives up and continues operation, the INKEY returning the value −1.

INKEY(N) thus performs a function rather similar to GET, except that GET will cause the computer to wait forever if necessary. The equivalent of GET$ is a closely related function, INKEY$, which returns a single character string variable instead of the ASCII value.

If what you really want is for the program to continue execution while checking for keyboard input, INKEY(0) will do this by checking the keyboard buffer but without any pause. Any program loop could contain, for example, the line

```
150 IF INKEY(0)>=0 THEN GOTO 200
```

which would accomplish this.

However, if a key had been pressed earlier, this would be picked up the first time round the loop. It is possible to 'flush' (i.e. empty) the keyboard buffer at any time before you enter the loop by

```
*FX15,1
```

to avoid this possibility.

Example 11.1

The following program uses the properties of INKEY(0) to give a simple reaction time test. The keyboard buffer is flushed at line 100 just before the loop detecting keyboard entry, to prevent cheating, then a REPEAT... UNTIL loop at lines 140 to 160 keeps track of time while checking for a key-press.

```
 10 CLS
 20 PRINT "When a message appears on the screen"
 30 PRINT '"press any key"
 40 TIME=0
 50 REPEAT: UNTIL TIME>400
 60 CLS
 70 PAUSE=INT(1000*RND(1))
 80 TIME=0
 90 REPEAT: UNTIL TIME>PAUSE
100 *FX 15,1
110 REM CLEARS KEYBOARD BUFFER
120 PRINT TAB(15,12);"PRESS NOW!"
130 TIME=0
140 REPEAT
150 T=TIME
160 UNTIL INKEY(0)>=0
```

```
170 PRINT '"You took ";T/100;" seconds"
180 PRINT '"Do you want to try again? (Y/N)"
190 YN$=GET$
200 IF YN$="Y" THEN GOTO 10
```

Since INKEY not only detects a key-press, but also returns its ASCII value, the program could be modified to look for a particular character selected at random. 64+RND(26) would generate the ASCII code for any capital letter, or more ambitiously 32+RND(94) would produce the ASCII code for *any* printable character.

Exercise 11.3

Modify the program above to choose a key at random, measure the reaction time and determine whether the user got it right.

Example 11.2

Another example of the use of INKEY is to temporarily freeze a display, of either text or graphics. In the next program, a series of sine curves are continuously plotted, but the display will be frozen by pressing any key, until a further key is pressed.

```
  10 MODE 4
  20 START=0
  30 REPEAT
  40 MOVE 0,500+400*SIN(RAD(START))
  50 FOR ANGLE=0 TO 360 STEP 5
  60 DRAW 3*ANGLE,500+400*SIN(RAD(ANGLE+START))
  70 IF INKEY(0)>=0 THEN PROC_wait
  80 NEXT ANGLE
  90 START=START-20
 100 UNTIL START=-360
 110 END
1000 DEF PROC_wait
1010 *FX 15,1
1020 REPEAT
1030 UNTIL INKEY(0)>=0
1040 ENDPROC
```

Line 70 is the key line, where INKEY is used within the plotting loop to check for a key-press, and if detected, call PROC_wait.

The third use of INKEY is with a negative argument. INKEY then checks whether a key is being pressed *at that instant*, returning −1 or zero according to whether the key is pressed or not. This form of the INKEY statement checks for a particular key, the key being determined by the value of the negative argument. The value of the number does not correspond to the key's ASCII code, but follows an obscure and seemingly random table which is given in Appendix I.

The most obvious application for this mode of INKEY is for games, where action must be taken according to which key has been pressed, but only for as long as the key is pressed. This mode of INKEY is not applicable to INKEY$ (it would be superfluous anyway).

All keys, including those such as CAPS LOCK, SHIFT, ESCAPE and the function keys, have a code. However, using keys such as ESCAPE or the arrow keys will give rise to a problem, because as soon as the INKEY command has been completed, when the user will almost certainly still have his finger on the key, the usual function will come into effect. It will again be necessary to turn off their normal functions with *FX 4,1 and *FX 229,1 as explained above.

Exercise 11.4

Modify the program of Example 11.2 so that instead of switching the plotting off and on with successive key presses, the display is frozen while the space bar is pressed (INKEY code −99) and plotting starts again as soon as the bar is released.

11.3 VALIDATION

11.3.1 String validation

The first aspect of validation is to ensure that, if the user presses one or more wrong keys the program does not 'crash' but instead tells the user that his/her input was wrong and re-poses the original question. Such an approach needs considerably more programming, but it is necessary if the program is to be safely used, particularly by people other than the original author.

Example 11.3

As an example of a helpful friendly style, the following would be an effective way to ask a simple yes/no question

```
  100 REPEAT
  110 PRINT "Random number = ";RND(1)
  120 PRINT "Do you want another number? (Y/N)"
  130 INPUT YN$
  140 IF YN$<>"Y" AND YN$<>"N" THEN PROC_wrongkey
  150 UNTIL YN$="N"
  160 END
 1000 DEF PROC_wrongkey
 1010 REPEAT
 1020 PRINT "Please press the 'Y' key for 'YES'"
 1030 PRINT "or the 'N' key for 'NO'"
 1040 INPUT YN$
 1050 UNTIL YN$="Y" OR YN$="N"
 1060 ENDPROC
```

Note that whichever key is pressed (apart from <BREAK>) the result should be intelligible to the user and the program will not crash.

The question requiring a yes/no response is actually a very common situation, but conventions vary as to whether the user responds with Y (N) or YES (NO). The program can be made to accept either response, (but also unfortunately YABOO or NUTS) by using INPUT with the string YN$, then testing the first character of YN$ for a Y or N. Try the following modifications to the above program

```
 140 IF LEFT$(YN$,1)<>"Y" AND LEFT$(YN$,1)<>"N" THEN
     PROC_wrongkey
 150 UNTIL LEFT$(YN$,1)="N"
1050 UNTIL LEFT$(YN$,1)="Y" OR LEFT$(YN$,1)="N"
```

11.3.2 Numerical validation

Because the number of 'valid' input strings is usually limited it is relatively easy to check them as above. Checking that a number is valid is more difficult. It is, however, worth mentioning some simple general methods.

In many cases a valid input number must lie within a known range. This can be easily tested with the <, > and = operations and a suitable warning printed if required. Similarly it is sometimes necessary for a number to be an integer. We can test if the number NUM is an integer as follows

```
    ...
    ...
100 REPEAT
110 INPUT "Type a number: ",NUM
120 IF NUM<>INT(NUM) THEN PROC_invalid
130 UNTIL NUM=INT(NUM)
    ...
    ...
```

where PROC_invalid prints out a suitable warning message.

On the BBC microcomputer, a value typed into a numerical variable in response to an INPUT command acts as if it were input as a string and converted to a numeric value by the VAL command (see Section 4.4). Although at first sight this appears to be a user-friendly technique, it suffers from the danger that there is no warning if a non-number key is accidentally pressed. The particularly prevalent mistake is to type the letter O instead of zero, for example 1O for 10, which would be converted to 1 with no warning, or perhaps more disastrously O.1 for 0.1 which would be converted to zero.

To avoid this possibility it is advisable to adopt a somewhat indirect approach to numerical input. You can pre-empt the operation of INPUT if the 'number' is initially input as a string variable and the function VAL is then used to convert it to an arithmetic variable (if the conversion is possible, otherwise VAL will give the result zero). Therefore, we can test whether the

STR$ of the arithmetic variable is equal to the original string and, if not, issue a warning and reprompt for input.

Example 11.4

A suitable piece of user-friendly program for the input of a number is

```
      ...
      ...
  100 REPEAT
  110 PRINT "Input a number"
  120 PROC_validate
  130 UNTIL TEST=1
      ...
      ...
  990 END
 1000 DEF PROC_validate
 1010 REM PROCEDURE TO VALIDATE NUMERICAL INPUT
 1020 LOCAL ST$
 1030 INPUT ST$
 1040 NUM=VAL(ST$)
 1060 IF ST$=STR$(NUM) THEN TEST=1: ENDPROC
 1070 PRINT "PLEASE DO NOT PRESS ANY NON-NUMBER KEYS"
 1080 PRINT "(apart from 'E' for a number"
 1090 PRINT "in exponent form)"
 1100 TEST=0
 1110 ENDPROC
```

These sections of program will assign the required number to the arithmetic variable NUM. The procedure can be used whenever a number is to be input (though it would be necessary to assign NUM to the required variable).

Exercise 11.5

Use the section of program above to input a number directly and print out its value and also its square. Investigate the effect of typing in a number and also a string including a letter (as if you had made a typing error). Delete lines 100 and 130 and replace line 120 by

```
120 INPUT NUM
```

and repeat the tests. In the latter case there will be no warning of the incorrect input.

One limitation of this kind of validation is that it is necessary for the number to be typed in in the same format as is adopted by the STR$ function for that particular number (i.e. exponent form for numbers outside the range 0.1 to 10^9). A much more complex checking process is required in order to detect any invalid input without imposing this restriction on the user, or

alternatively using INPUT directly with a number and risking incorrect input. The next subroutine will accept almost any valid input, but trap any error.

Example 11.5

PROC_validate in the program below will validate the input of a number and return with the number stored in the variable ZQ. This can then be assigned to any desired variable. The program itself serves merely to give you an opportunity to test out the procedure.

```
   10 REPEAT
   20 PRINT "INPUT A NUMBER"
   30 PROC_validate
   40 PRINT "YOUR NUMBER IS: ";ZQ
   50 UNTIL FALSE
   60 END
10000 DEF PROC_validate
10010 REM PROCEDURE TO VALIDATE THE INPUT OF A NUMBER
10020 REM THE NUMBER IS RETURNED IN VARIABLE ZQ
10030 REM DPTS COUNTS DECIMAL POINTS
10040 REM EXN    "    EXPONENT E'S
10050 REM ST MARKS THE START OF NUMBERS
10060 REM NMT COUNTS THE POWER OF 10 OF THE MANTISSA
10070 REM INP$ IS THE NUMBER
10080 REM EX$ IS THE EXPONENT
10090 REM CH$ IS A SINGLE CHARACTER OF THE NUMBER
10100 LOCAL J,DPTS,EXN,ST,NMT,INP$,NUM$,CH$,CANC$,LST$
10110 INP$="": NUM$="": DPTS=0: EXN=0: ST=0: NMT=0
10120 PROC_inchar
10130 PRINT
10140 ZQ=VAL(INP$)
10150 ENDPROC
11000 DEF PROC_inchar
11010 CH$=GET$
11020 IF CH$=CHR$(13) THEN ENDPROC
11030 IF CH$=CHR$(127) THEN PROC_cancel: ENDPROC
11040 PRINT CH$;
11050 IF ((CH$<"0" OR CH$>"9") AND CH$<>"." AND CH$<>"-" AND
      CH$<>"+" AND CH$<>"E" AND CH$<>CHR$(13)) THEN
      PROC_invalid: ENDPROC
11060 IF (CH$="-" OR CH$="+") AND ST=1 THEN PROC_invalid:
      ENDPROC
11070 IF EXN>0 OR DPTS>0 OR CH$<"0" OR CH$>"9" THEN ELSE IF
      NMT<=37 THEN NMT=NMT+1 ELSE PROC_invalid: ENDPROC
11080 IF EXN=0 OR CH$<"0" OR CH$>"9" THEN ELSE IF
      (VAL(NUM$+CH$)+NMT)<=38 THEN NUM$=NUM$+CH$ ELSE
      PROC_invalid: ENDPROC
```

```
11090 IF CH$<>"." THEN ELSE IF DPTS=0 AND EXN=0 THEN DPTS=1
      ELSE PROC_invalid: ENDPROC
11100 ST=1
11110 IF CH$<>"E" THEN ELSE IF EXN=0 THEN EXN=1: ST=0: ELSE
      PROC_invalid: ENDPROC
11120 INP$=INP$+CH$
11130 PROC_inchar
11140 ENDPROC
12000 DEF PROC_invalid
12010 PRINT CHR$(7);CHR$(127);
12020 PROC_inchar
12030 ENDPROC
13000 DEF PROC_cancel
13010 IF LEN(INP$)=0 THEN VDU 7: PROC_inchar: ENDPROC
13020 PRINT CH$;: CANC$=RIGHT$(INP$,1)
13030 INP$=LEFT$(INP$,LEN(INP$)-1)
13040 IF LEN(INP$)=0 THEN LST$="" ELSE LST$=RIGHT$(INP$,1)
13050 IF LST$="E" OR LST$="" THEN ST=0 ELSE ST=1
13060 IF CANC$="." THEN DPTS=0
13070 IF CANC$="E" THEN EXN=0
13080 IF CANC$="+" OR CANC$="-" THEN ST=0
13090 IF EXN=0 AND DPTS=0 AND CANC$>="0" AND CANC$<="9" THEN
      NMT=NMT-1
13100 IF EXN>0 AND CANC$>="0" AND CANC$<="9" THEN
      NUM$=LEFT$(NUM$,LEN(NUM$)-1)
13110 PROC_inchar
13120 ENDPROC
```

Note the use of recursion, and also the use of null statements between THEN and ELSE to avoid the ambiguity of IF...THEN...IF...THEN...ELSE.

11.4 DISPLAY LAYOUT

A well thought out and attractive presentation of text is necessary for programs which are to be pleasant and interesting to use. BBC BASIC includes several useful commands to control the presentation of text.

11.4.1 Layout of text

The layout of numbers is more complicated than that of text, so we will deal with text first.

All screen character output takes place at the cursor position, and the cursor can be positioned prior to printing in a variety of ways. The simplest is by means of the alternative PRINT list separators, the semicolon (;), comma (,) and apostrophe (').

The effect of the apostrophe is simplest. It generates a new line and returns the cursor to the left of the screen. It is the one item of punctuation in

BBC BASIC which is not standard in other versions of BASIC. It has been used fairly extensively in programs listed in this book, particularly to avoid repeated PRINT statements. Thus

```
PRINT '"LEAVE A SPACE"'"NEW LINE"
```

is equivalent to

```
PRINT: PRINT: PRINT "LEAVE A SPACE": PRINT "NEW LINE"
```

The effect of the semicolon is to separate items which are to be printed consecutively, with no intervening spaces. For instance

```
PRINT "DIAL";999;"FOR POLICE"
```

This appears as

```
DIAL999FOR POLICE
```

with no spaces. In order to space it out properly, it is essential to put spaces inside the quotation marks

```
PRINT "DIAL ";999;" FOR POLICE"
```

```
PRINT "DIAL"; 999; "FOR POLICE" will not do
```

Note that very often it is unnecessary to include the semicolons. Whenever the computer can make sense of the print list without them, it will do so. Generally speaking, almost anything but numbers or real variables can be PRINTed without intervening semicolons. The problem with numbers and real variables is that they simply merge into one another. Even string and integer variables, on the other hand, can be distinguished by the $ or % at the end.

Thus we could put

```
PRINT "DIAL "999" FOR POLICE"
```

Similarly

```
A$="STRINGING "
B$="THINGS "
C$="TOGETHER"

PRINT A$B$C$
```

will produce the desired message, because the computer recognizes the dollar signs as denoting the end of each string variable.

However, it is not always possible to leave the semicolons out. Try the effect of

```
PRINT "STRINGING ""THINGS ""TOGETHER"
```

Two consecutive quotation marks are used to generate an actual quotation mark within a string. (They are also needed at the *start* of a DATA item or a string response to INPUT, because these may optionally be enclosed in quotation marks – for example, if a comma is to be included in the string. See Section 4.2.3.)

Generally speaking, it is best to develop the good habit of always including the semicolon.

One extra feature of the semicolon is that, if used at the end of a PRINT statement, it suppresses the newline, and leaves the cursor immediately after the printed text, which will enable you to carry on printing at a later stage if you wish. Try

```
PRINT "PUTTING THINGS";: PRINT "TOGETHER"
```

Also try just

```
PRINT "SO FAR SO GOOD";
```

The comma serves quite a different function. It behaves exactly like a preset tabulator key on a typewriter, so that each comma moves the cursor to the next available tab position across the screen. Normally the tab positions are ten characters apart. Try

```
PRINT "ME","AND YOU","AND US","."

PRINT "TOO LONG FOR ONE","TAB POSITION"
```

Ending a PRINT statement with a comma will *not* cause the next PRINT to occur on the same line.

The default 'tab setting' of 10 characters can be changed. This value forms the first byte of the resident integer variable @%, so changing this, for example to

```
@%=@%+5
```

will increase the spacing to 15 characters.

Exercise 11.6

Experiment with layouts of strings while altering the value of @%.

11.4.2 TAB, SPC and POS

More elaborate layouts are possible with various combinations of the PRINT commands TAB and SPC, and the function POS.

The TAB command has two forms. The simpler is TAB(X), which when used within a PRINT statement moves the cursor along to column X in the line. It thus acts like an adjustable tabulator, giving much better control over layout. A few points should be noted about TAB. The range of X is 0 to 255. Larger numbers are treated modulo 256 (as are negative numbers). Non-integer numbers are truncated (not rounded). Numbers larger than the screen width will cause wrap round for as many lines as necessary. Also, if column X has been passed, wrap round will occur to column X on the next line. This last point is a potential source of danger.

Note that the left hand column of the screen is column 0, *not* 1, and thus on a 40 column display the right hand column is 39. This is actually fairly logical, since PRINT TAB(5);"STRING" will generate 5 spaces before STRING.

Another way in which output can be controlled is with the SPC(N) command, which must be used in the same way as TAB(X) within a PRINT statement. SPC(N) will move the cursor on N spaces, wrapping round if necessary. The value of N follows the same rules as for X in TAB, so the effective range is 0 to 255.

A useful function in conjunction with TAB and SPC is POS, which returns the current column in which the cursor lies. This could be used with TAB to check that the cursor had not passed the intended TAB position, or with SPC to produce an effect similar to TAB. It will be appreciated that there is a large element of duplication in these functions, giving the programmer the choice of his preferred technique, as is emphasized by the following exercise.

Exercise 11.7

Try the following statements which all give an equivalent layout

```
PRINT "OTHER,A.N.(type 5 spaces)01-234-5678(type 4 spaces)LONDON"
PRINT "OTHER,A.N.";SPC(5);"01-234-5678";SPC(4);"LONDON"
PRINT "OTHER,A.N.";TAB(15);"01-234-5678";TAB(30);"LONDON"
PRINT "OTHER,A.N.";SPC(15-POS);"01-234-5678";SPC(30-POS);
      "LONDON"
```

Exercise 11.8

Write a procedure, PROC_table, to print out as a table the string array A$ having DIMension A$(10,5), in the format of 10 rows of 5 columns, each 7 characters wide. The table should be headed with the string array HEAD$ (DIMension HEAD$(5)), which should be separated from the body of the table by two blank lines. Use it in conjunction with the following program.

```
10 DIM A$(10,5),HEAD$(5)
20 CLS
30 FOR J=1 TO 5: HEAD$(J)="COL"+STR$(J): NEXT J
40 FOR J=1 TO 5
```

```
 50 FOR K=1 TO 10
 60 A$(K,J)=STR$(K)+STR$(J)
 70 NEXT K
 80 NEXT J
 90 PROC_table
100 END
```

11.4.3 TAB(X,Y)

The second, and much more powerful, use of TAB is in the form TAB(X,Y), with two parameters which give the horizontal and vertical positions on the screen to which the cursor is to be moved. As with TAB(X), the left hand column is column zero, and the top row (not the bottom) is row zero.

TAB(X,Y) will move the cursor immediately to any point on the screen, without printing intervening spaces as TAB(X) does. Thus TAB(X,Y) does not overwrite the earlier part of the line, or indeed anywhere else on the screen. The whole philosophy of output to the screen should be altered when using TAB(X,Y), so that instead of printing lines successively down the screen, output is directed and redirected to any point on the screen.

If either parameter of TAB (X,Y) is out of range (outside the values 0 to 39 for X and 0 to 24 for Y in Mode 7, for example), the command is ignored completely, in contrast to TAB(X).

Example 11.6

As an example of the use of TAB(X,Y), the program below prints out a string forwards and then backwards.

```
 10 REM PROGRAM TO ILLUSTRATE TAB(X,Y)
 20 CLS
 30 A$="A MAN A PLAN A CANAL PANAMA"
 40 L=LEN(A$)
 50 LFT=INT(19-L/2)
 60 FOR J=1 TO L
 70 PRINT TAB(LFT+J-1,8);MID$(A$,J,1)
 80 PRINT TAB(LFT+L-J,16);MID$(A$,L+1-J,1)
 90 TIME=0
100 REPEAT: UNTIL TIME=20
110 NEXT J
120 PRINT TAB(0,24);
```

If you have the supplementary disc which is available to accompany this book, program TABDEMO shows a more sophisticated use of TAB(X,Y), for an attractive method of input of data to set up a simple database.

Exercise 11.9

If you have the supplementary disc, program CLOCK on the disc allows you to set a time and then provides the time as the variables HOUR1, MIN1 and

SEC1. It also provides a stopwatch function so that pressing 1 starts the watch, 2 stops it and 3 resets it. The stopwatch time is provided in HOUR2, MIN2 and SEC2. Write a procedure, PROC_clock, needed for program CLOCK, to display the time and the stopwatch readings on the centres of lines 8 and 16 respectively.

Corresponding to the function POS, which is useful in conjunction with TAB(X) and SPC, there is a second function VPOS which returns the vertical position of the cursor. Thus to move to the start of the current line, you might use

```
PRINT TAB(0,VPOS);
```

After using TAB(X,Y) in a PRINT statement, note that TAB(X) and commas will be reset to measure from the position of the cursor after the TAB(X,Y) call, instead of the left-hand edge of the screen.

11.4.4 Output to a printer

On the BBC microcomputer, the layout of printer output is quite simple, because the functions TAB(X), SPC and POS all work for printer as well as screen output. TAB(X,Y) of course cannot possibly work with printers, and is ignored. Moving printer paper backwards requires special codes particular to each printer, so all printed output should be planned to move continuously down the page.

There are two extra commands, WIDTH and COUNT, which affect both screen and printer output, but are probably most relevant to the latter.

WIDTH N

Sets the *page width* to N characters. It can thus be used to limit the screen display (especially perhaps for Mode 0), but it also generates a newline code and so could be particularly useful to limit the width of printer output. For example

```
WIDTH 72
```

would restrict an 80 column printer to 72 columns, which would be a more suitable width for A4 paper.

COUNT

Is a function returning the number of characters output since the last newline. Within one screenwidth it is equivalent to POS, but POS relates only to the cursor position on the screen. Thus COUNT should be used rather than POS for printer output, where wrap-round may have occurred on the screen.

11.5 FORMATTING NUMERICAL OUTPUT

11.5.1 Layout of numbers

All of the previous discussion of layout of text also applies to numbers, that is the printing of real or integer variables. In addition, however, there are

considerations of field width, significant figures, format and mode of display.

The first point to notice is that numbers are normally printed right justified. Thus if we issue the command

```
PRINT 12345,67890
```

the result is

```
.....12345.....67890
```

(Note that the dots do not appear on the screen, but indicate the number of spaces between the characters printed.) The effect of the semi-colon is not only to act as a separator in the PRINT list, but also to cancel this right justification. Try

```
PRINT ;12345,67890
```

The reason for the right justification is so that integer numbers of different lengths, printed in a column, all have their units column aligned. Try

```
PRINT 1: PRINT 23: PRINT 456
```

However, real numbers with decimal fractions cannot be aligned this way unless they all have the same number of decimal places. Try

```
PRINT 1.2: PRINT 3.4: PRINT 5.67
```

If you want to retain this right justification while tabulating with TAB(X) or TAB(X,Y) rather than commas, you must omit the semicolons which good practice would normally dictate that you should include. Try

```
PRINT 1 TAB(15) 23: PRINT 45 TAB(15) 678
```

then

```
PRINT 1;TAB(15);23: PRINT 45;TAB(15);678
```

On the other hand, if you want numbers printed with spaces in between, then you must include the semicolons. Compare

```
PRINT 12" "34" "56 and PRINT 12;" ";34;" ";56
```

11.5.2 Controlling the format of printed numbers

It is often desirable to have a close control over the format of printed numbers. Two very common requirements are to fix the number of decimal places displayed, and to print decimal numbers in columns such that the decimal points are properly aligned.

To a limited degree these features can simply be achieved with the aid of

the resident integer variable @%. To print all numbers with N decimal places, in columns of width W, set

```
@%=&20000+&100*N+W
```

Example 11.7

The program below generates numbers whose magnitudes and number of figures, as well as values, are random, and prints them with 2 decimal places in 3 columns of width 10 characters. This would for instance be very suitable for displaying columns of money in pounds.

```
  10 CLS
  20 @%=&2020A
  30 FOR ROW=1 TO 20
  40 FOR COL=1 TO 3
  50 REPEAT
  60 NUM=FNrannum
  70 UNTIL ABS(NUM)<1E5 AND ABS(NUM)>0.1
  80 PRINT ,NUM;
  90 NEXT COL
 100 PRINT
 110 NEXT ROW
 120 END
1000 DEF FNrannum
1010 LOCAL DECPLACES, NUM
1020 DECPLACES=10^(RND(6)-1)
1030 NUM=INT((RND(1)-0.5)*10^RND(9))
1040 =NUM/DECPLACES
```

It would be as well to press <BREAK> after using this program, to restore the default value for @%.

The variable @% can do a great deal more than just produce a fixed number of decimal places, but its use is quite complicated, and is therefore described in Appendix F.

The biggest limitation of the example above is that all columns must be of equal width and, moreover, the decimal places are displayed even for integers. We might, however, want a column of 3 figure integers, two columns of numbers of up to 7 significant figures, having two decimal places, then two columns of numbers less than one, having just 3 decimal places. What is needed is a way in which successive numbers can be printed in different formats.

Example 11.8

The following procedure will make it easy to print numbers with close control over their format. It prints the number NUM, starting from the cursor position on the current line, with up to N decimal places and a field width of W (or more, if the number cannot be accommodated in that number of characters). The cursor is left at the end of the number. Numbers are justified

within the field width so that the decimal point is N places from the right of the field, and only significant figures are printed. If N is set to 0, numbers are printed as integers. (The procedure will not cope with numbers larger than 10^9 or smaller than 0.1 that need to be in exponent form.)

```
10000 DEF PROC_formprint(NUM,N,W)
10010 REM PROCEDURE TO PRINT NUM
10020 REM WITH A FIELD WIDTH OF W
10030 REM AND N DECIMAL PLACES.
10040 LOCAL ST1$,ST2$,ST3$,ST4$,ST5$,M,J,DPPOS
10050 @%=@% OR &1000000
10060 ST1$=" "+STR$(NUM)
10070 ST2$=" "
10080 ST3$="": ST4$=""
10090 ST5$="."
10100 IF NUM=INT(NUM) THEN ST1$=ST1$+". ":ST5$=" "
10110 M=W-N+(N>0): REM M IS MINIMUM NUMBER OF FIGURES BEFORE
      DECIMAL POINT
10120 DPPOS=0
10130 REPEAT
10140 DPPOS=DPPOS+1
10150 UNTIL MID$(ST1$,DPPOS,1)="." OR DPPOS=LEN(ST1$)
10160 IF DPPOS>M THEN ST3$=LEFT$(ST1$,DPPOS-1): GOTO 10210
10170 ST3$=LEFT$(ST1$,DPPOS-1)
10180 FOR J=DPPOS TO M
10190 ST3$=ST2$+ST3$
10200 NEXT J
10210 ST4$=RIGHT$(ST1$,LEN(ST1$)-DPPOS)
10220 FOR J=1 TO N
10230 ST4$=ST4$+ST2$
10240 NEXT J
10250 PRINT ;ST3$;
10260 IF N>0 THEN PRINT ;ST5$+LEFT$(ST4$,N);
10270 ENDPROC
```

Adapt the program from Example 11.7 to generate numbers with the same random format as before, and print them using the above procedure, by deleting line 20 and replacing line 80 by

```
80 PROC_formprint(NUM,4,13)
```

Add the procedure above and test out the program. You will probably find this procedure easier to use than @% as described in Appendix F.

Exercise 11.10

Write a program to use PROC_formprint to print numbers (generated at random) in the format suggested earlier, that is one 3 figure integer, two 7 figure numbers and so on.

In general use of this procedure, it is important to make W large enough for all possible numbers that might occur (and perhaps check their size before printing).

11.6 GENERAL PRESENTATION

There are several features that will generally add to the presentation of the output of a program. The most obvious is that you should always clear the screen at the start, either with CLS or, better still, a MODE command which will also ensure that you are in the desired mode.

Another small point is that whenever you want to attract the user's attention, for instance when prompting for input or warning of a mistake in entering a value, use the 'bell' character, either with PRINT CHR$(7) or VDU 7.

As well as using sound to attract a user's attention, it is possible to emphasize text in various ways, to draw attention to it or simply to make a presentation more attractive.

When we consider the ways of producing attractive presentation we must distinguish Mode 7, which is different from all the other modes.

11.6.1 Mode 7

The method of producing special displays in Mode 7 is quite unlike that in other modes. These are not produced by BASIC commands such as COLOUR, but by ASCII codes printed (invisibly) on the screen. Only predefined characters and shapes can be used, at a fixed set of character positions, and colours are fixed for a whole character position at a time.

The text display consists of 25 lines of 40 characters, making 1000 character cells, and all the screen control is through the contents of these cells. If a cell contains one of the special ASCII codes controlling the display, it modifies the way in which the characters to the right of the code on the same line are interpreted.

Try the following commands in Mode 7.

```
PRINT "ABCD";CHR$(129);"efgh";CHR$(130);"IJKL"
```

ASCII codes 129 and 130 alter the colour of the rest of the line to red and green respectively. Notice that the codes each occupy a character cell. It is not possible to produce a continuous line of text, such as

```
ABCDefgh
```

with, say, the lower case letters of different colour to that of the capitals.

The same range of steady colours is available in Mode 7 as in Mode 2 (with the exception of black), and the codes that produce them are shown in Table 11.1.

Table 11.1 Special display codes for Mode 7.

		Special effects	
Colour	*Code*	*Code*	*Purpose*
red	129	136	flash on
green	130	137	flash off
yellow	131	141	double height characters
blue	132	140	normal characters
magenta	133	157	new background colour
cyan	134	156	black background
white	135	152	conceal display

It is possible to make the colours flash, but unlike other modes the flash is between the single colour and the background, and the on/off periods are of unequal duration.

As we have seen before, there are two ways to produce ASCII codes. One way is to use CHR$() from within a PRINT statement. The second way, which is more convenient when a string of ASCII codes is to be issued, as will sometimes be necessary, is via the VDU command. All the values following VDU are issued as ASCII codes. In other display modes, VDU is used as the most convenient way to issue low value ASCII control codes which require parameters, but in Mode 7 it will equally well issue the high values needed.

There is actually a third way to produce the display control codes. Pressing <SHIFT> and a function key simultaneously will produce the character with ASCII code of 128 plus the function key number. This is primarily designed for use in direct mode, but the characters can be embedded in PRINT strings to produce, for example, coloured text from within a program. Try

```
PRINT "ABCD<SHIFT-f1>efgh<SHIFT-f2>IJKL"
```

This method has the additional advantage that the colours show up when the program is listed in Mode 7, but watch out for the fact that the codes will not be apparent in a printer listing of the program.

Just as in the other modes, it is possible to change the background from black to a colour, or back again to black. Code 157 must be issued *after* the code of the background colour you want. You will then need to issue a new colour code for the foreground text, or it will be invisible. For example, to print in green on a red background you would need

```
VDU 129,157,130: PRINT "THIS MAKES ME SEE RED"
```

This is where the VDU command becomes particularly useful.

You will see that the above line creates a complete background line in red. You could prevent this by finishing with code 156, which causes the

background to revert to black. However, the leading 'spaces' of the two codes cannot be prevented from appearing in red. Try

```
VDU 129,157,130: PRINT "ON AND OFF";: VDU 156: PRINT
```

Another special effect in Mode 7 is the production of double height (but normal width) characters, which is initiated by code 141 (and switched off, if necessary later in the line, by 140). An unexpected feature, but which can be understood from the strict way that Mode 7 works with character cells only, is that double height letters must be printed twice on successive lines, both lines being identical (including code 141 at the start). Try the effect out with lines such as

```
PRINT CHR$(141);"A BIG MESSAGE": PRINT CHR$(141); "A BIG
MESSAGE"
```

Try also the effect of disobeying the rules, with for instance

```
PRINT CHR$(141);"NOT A": PRINT CHR$(141);"MATCH"
```

There are four obvious ways when using Mode 7 to emphasize a section of text.

1. Use coloured text.
2. Use double height text.
3. Use flashing text. This is particularly eye catching, and can be used to tell the user what to do – for example

   ```
   PRINT CHR$(136);"Press space bar to continue"
   ```

   In other contexts it should be used in moderation or it can become overpowering.
4. To print text on a white or coloured background (unlike other modes, it is not possible to have black on a coloured backgound).

The following program illustrates all these techniques.

Example 11.9

```
10 MODE 7
20 CLS
30 DIM DY$(7)
40 FOR I=1 TO 7: READ DY$(I): NEXT I
50 PRINT TAB(10,6);CHR$(130);CHR$(141);"SPELLING TEST"
   'TAB(10);CHR$(130);CHR$(141);"SPELLING TEST"
60 PRINT TAB(10,9);CHR$(131);"Days of the week"
70 FOR I=1 TO 10
```

```
 80 DAY=RND(7)
 90 PRINT TAB(0,12);"What is the name of day ";DAY; " of the
    week? "
100 PRINT SPC(14);: INPUT DAY$
110 PRINT TAB(0,12);SPC(80)
120 IF DAY$=DY$(DAY) THEN PRINT TAB(13,12);CHR$(129);
    CHR$(136);"CORRECT": SC=SC+1: ELSE PRINT TAB(0,12);
    CHR$(132);"Day ";DAY;" of the week is spelt:-";
    CHR$(136);DY$(DAY)
130 PRINT TAB(5,23);CHR$(136); CHR$(134);"Press any key to
    continue": A$=GET$
140 PRINT TAB(0,23);SPC(40);
150 NEXT I
160 CLS
170 IF SC>=8 THEN PRINT TAB(10,10);CHR$(141);
    "CONGRATULATIONS"'TAB(10);CHR$(141);"CONGRATULATIONS"
180 PRINT TAB(7,15);CHR$(133);"You scored ";SC;" out of 10"
190 DATA SUNDAY,MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,
    SATURDAY
```

11.6.2 Modes 0 to 6

With Modes 0 to 6 almost unlimited effects are possible, but this would generally involve the use of graphics. With Modes 1, 2 or 5 dramatic effects with colour are also possible. The simple commands available for graphics, plus related commands also applying to text, are dealt with in Chapter 6. We shall therefore deal with just one technique, which can be used in *all* modes – printing in inverse, that is black on a white background. If you wished, it should not be difficult to modify the procedure to print in one colour on a differently coloured background.

Example 11.10

The following procedure will print in inverse the string or string variable TEXT$, in any mode (even Mode 7, but here the text has to be in blue, because you cannot print in black in this mode).

```
10000 DEF PROC_inverse(TEXT$)
10010 LOCAL WHITE
10020 A%=135
10030 MD=USR(&FFF4)
10040 MD=(MD AND &FFFFFF) DIV &10000
10050 IF MD=7 THEN GOTO 10130
10060 WHITE=1
10070 IF MD=1 OR MD=5 THEN WHITE=3
10080 IF MD=2 THEN WHITE=7
10090 COLOUR 128+WHITE: COLOUR 0
10100 PRINT TEXT$;
```

```
10110 COLOUR 128: COLOUR WHITE
10120 ENDPROC
10130 VDU 157,132: PRINT TEXT$;
10140 VDU 156,135
10150 ENDPROC
```

Exercise 11.11

Use PROC_inverse above to enhance one of the earlier programs, or a program of your own, with inverse text.

11.7 MENU DRIVEN PROGRAMS

A useful feature of a disc-based BBC microcomputer is its ability to function as a 'turnkey' system – which automatically loads and runs the required program when <SHIFT-BREAK> is pressed. The method of doing this was explained in Section 8.5. If a turnkey system is simply required to run the main program on the disc, then the !BOOT file will probably be a *EXEC ASCII file containing the single command CHAIN "<main program name>".

The idea of a turnkey system is that it should make it as easy as possible for a complete non-programmer to get the program on the disc up and running. Very often, a disc will contain more than one program; the standard way of allowing the user to select the particular program that he wants is by means of a *menu*. A list of the available programs is presented to the user, like a restaurant menu, and the user selects one by a single key – usually either a number or letter. Instead of the main program on the disc, the !BOOT file must CHAIN a preliminary menu program.

As an example, suppose that a disc contains 3 main programs

1. PAY – a program that deals with the company payroll.
2. TAX – a program that deals with tax liabilities.
3. BONUS – a program which deals with the company's bonus and incentives scheme.

Example 11.11

The program (MENU) below shows a suitable preliminary menu program for a menu driven turnkey system.

```
10 REM MENU PROGRAM
20 CLS
30 DIM P$(3)
40 P$(1)="PAY"
50 P$(2)="TAX"
60 P$(3)="BONUS"
70 PRINT "The programs available on this disc"
80 PRINT '"deal with:"
90 PRINT '"1)  The company payroll"
```

```
100 PRINT '"2)  Tax liabilities"
110 PRINT '"3)  The company's bonus and incentive"
120 PRINT "scheme"
130 PRINT '""Press 1, 2 or 3 for your choice"
140 A$=GET$
150 IF VAL(A$)<1 OR VAL(A$)>3 THEN PRINT "YOU MUST PRESS KEYS
    1, 2 OR 3": GOTO 140
160 CHAIN P$(VAL(A$))
```

It is frequently convenient in a turnkey system for the last executed statement of the programs on the disc (PAY, TAX and BONUS in the above example) to bring the user directly back to the menu by rerunning the menu program. The command

```
CHAIN "MENU"
```

will achieve this.

Example 11.12

A more elaborate menu, which is another example of the use of attractive presentation, is given in the program below.

```
  5 *TV0,1
 10 MODE 1
 20 DIM PROG$(6)
 30 PROG$(1)="PAY"
 40 PROG$(2)="TAX"
 50 PROG$(3)="BONUS"
 60 PROG$(4)="PROFIT"
 70 PROG$(5)="BALANCE"
 80 PROG$(6)="INVESTMENT"
 90 PRINT TAB(0,3);"This disc contains programs for the"
100 PRINT '"company's financial affairs"
110 PRINT
120 STARS$=STRING$(40,"*")
130 PRINT STARS$;
140 GAP$="*"+STRING$(38," ")+"*"
150 PRINT GAP$;
160 PRINT "*   1) The company payroll            *";
170 PRINT GAP$;
180 PRINT "*   2) Tax liabilities                *";
190 PRINT GAP$;
200 PRINT "*   3) Bonus and incentive scheme     *";
210 PRINT GAP$;
220 PRINT "*   4) Profit and loss account        *";
230 PRINT GAP$;
```

```
  240 PRINT "*   5) Balance sheet                    *";
  250 PRINT GAP$;
  260 PRINT "*   6) The company's investments        *";
  270 PRINT GAP$;
  280 PRINT "*   7) END                              *";
  290 PRINT GAP$;
  300 PRINT STARS$
  310 PRINT "Press: ";
  320 PROC_inverse("<SPACE BAR>")
  330 PRINT " to move up and down"
  340 PRINT "                    the list"
  350 PRINT 'SPC(7);
  360 PROC_inverse("<RETURN>")
  370 PRINT " to run required program"
  380 VDU 23;8202;0;0;0;: REM TURN OFF CURSOR
  390 VDU 19,1,8;0;: VDU 19,130,143;0;
  400 PROGNUM=2: DIRN=-1: CURS=11
  405 PROC_arrow
  410 REPEAT
  420 CH$=GET$
  430 IF CH$=" " THEN PROC_arrow
  440 UNTIL CH$=CHR$(13)
  450 MODE 1: REM ALSO RESTORES CURSOR
  460 IF PROGNUM<7 THEN PRINT TAB(7,15);:PROC_inverse
      ("Loading requested program"): B%=PROGNUM: C%=1: CHAIN
      PROG$(PROGNUM)
  470 END
10000 DEF PROC_inverse(TEXT$)
10010 COLOUR 129: COLOUR 0
10020 PRINT TEXT$;
10030 COLOUR 128: COLOUR 1
10040 ENDPROC
20000 DEF PROC_arrow
20010 PRINT TAB(2,CURS);"  ";
20020 IF PROGNUM=1 OR PROGNUM=7 THEN DIRN=-DIRN
20030 PROGNUM=PROGNUM+DIRN
20040 CURS=2*PROGNUM+7
20050 COLOUR 1: COLOUR 130
20060 PRINT TAB(2,CURS);"->";
20070 COLOUR 4: COLOUR 128
20080 ENDPROC
```

Exercise 11.12

Write a !BOOT file and menu program so that, when <SHIFT-BREAK> is pressed, the programs on the disc are automatically catalogued. Then the user is asked which program is to be run, the answer is loaded into a string P$, say, and that program is CHAINed.

Exercise 11.13

Write a menu driven program to carry out one of a set of possible mathematical operations on a provided argument. The program should first use GET to choose from the following list of operations on a number N

reciprocal of N
ten to power N
factorial of N
PI to N decimal points
N random numbers

Use a second GET to request a single number argument for the operation, with an explanatory request.

(Factorial N can be obtained from PROD=1: FOR J=1 TO N: PROD= PROD*J: NEXT J).

Appendix A

SPECIAL FEATURES OF THE BBC MICROCOMPUTER

In this appendix we shall briefly review all the features particular to the BBC microcomputer which have not been specifically dealt with elsewhere.

THE KEYBOARD

The BBC microcomputer keyboard includes a full 96 character ASCII keyboard, including lower case letters, although one symbol (ASCII code 96) is used as the pound sign instead of the grave accent (`), which is the standard symbol. It has both CAPS LOCK and SHIFT LOCK keys, the former giving upper-case letters but the lower-case option on all the special symbol keys. This is the default option when the computer is switched on, because it is most appropriate for programming, where upper case *must* be used for all BASIC keywords. Both CAPS LOCK and SHIFT LOCK are toggle switches rather than press and release keys as on a normal typewriter. When either key is engaged, this is shown by an illuminated LED below the key. (N.B. When both LEDs are illuminated together, this is quite a separate function, indicating that computer output is suspended. This can happen either when listing to the screen in paged mode, or during output to a printer which has a full buffer.)

A number of the keys have the standard ASCII control code functions

```
CTRL  RETURN  ESCAPE  DELETE  TAB
```

The BREAK key is the most powerful key on the computer. Unless disabled, it will interrupt and destroy the operation of any program, no matter in what state, and will clear the computer of most hung states. It clears the program, although it can normally be recovered with the command OLD, and puts the computer back into Mode 7 restoring many defaults. An even more powerful <BREAK> is obtained when the CTRL key is depressed in conjunction with the BREAK key, and this combination cannot be easily disabled. <BREAK> has yet other effects in combination with other keys, in particular the SHIFT key; <SHIFT-BREAK> is used to auto-boot the disc or Econet filing system.

When <BREAK> is pressed in conjunction with any other key, such as <SHIFT> or <CTRL>, it is necessary to keep the other key pressed while the BREAK key is released first. The reason for this is that while <BREAK> is pressed, the microcomputer is completely disabled. When <BREAK> is released, the microcomputer reads the keyboard to see whether another key is being pressed, and acts according to which key it is, if any.

The five cursor control keys (the four arrow keys plus the COPY key) on the right hand side of the keyboard are used for screen editing. Pressing any one of the arrow keys invokes a second cursor, the copy cursor, which can then be moved anywhere on the screen (with wrap-round) by the arrow keys. Wherever the copy cursor is on the screen, the COPY key can be used to 'pick up' the character at that point and build it into the current line of input. Thus a program line with minor errors can easily be edited without complete re-entry, by copying the old line in this way, while making any necessary corrections by typing in or deletion as necessary.

OPERATING SYSTEM COMMANDS

It is important from the start to distinguish two types of commands that can be issued to the BBC computer. Most commands, such as PRINT, MODE, VDU and CLS, are BASIC commands and are handled by the BASIC interpreter.

There is another set of commands that is handled directly by the Machine Operating System (MOS for short, or just OS). All of these commands begin with a star, and this tells the BASIC interpreter that the command that follows is to be passed directly to the OS. If the OS does not recognize the command it is passed on to any paged ROMs present in the computer, starting at the right-hand end of the bank of ROMs. Most such commands are likely to be intended for the filing system ROM. If none of the ROMs claim the command, it is treated as an abbreviation of the *RUN command, and any file in the filing system having the same name as that following the star will be executed as a machine code program. To further emphasize the difference between BASIC and Operating System commands, note that the latter can be handled even if the computer does not possess a BASIC ROM, or is currently operating in a different language, such as *Wordwise* or Pascal.

Two points should be noted about Operating System commands. First, because the command is passed on to the Operating System, no BASIC commands can follow an OS command on the same line.

Second, OS commands can be issued either in upper case, like BASIC commands, or lower case. This includes filenames used for any of the filing systems; the name will appear in any listing in the form that was used when the file was created, but it will be recognized subsequently whether in upper or lower case.

THE USER-DEFINED FUNCTION KEYS

The most eye-catching keys on the keyboard are the set of 10 red keys along the top, marked f0 to f9. These are called the user-defined function keys,

because they can be programmed by the user to carry out any function he wishes. When a function key is pressed, the contents programmed into the key are issued to the computer just as if they had been typed in at the keyboard. This can be very usefully employed to issue frequently used commands, or even that part of a command which does not alter.

The command to program the key marked fn (where n=0 to 9) is

```
*KEY n <any text you wish to be issued> <RETURN>
```

For instance, you could set up key 0 to issue the word RENUMBER at a single keystroke, after which you could add the starting line number and increment and press <RETURN>. The command needed would be

```
*KEY 0 RENUMBER <RETURN>
```

Suppose, however, that you are happy with the default values for RENUMBER. It still requires two keystrokes to complete the command because you have to press the <RETURN> key. This key cannot be directly programmed into the function key, because it is already used to terminate the programming line for the key. However, there is a way of including a <RETURN> key code, or any other control code, into the function keys. You must type the symbol || (the one over the backslash at the top right of the keyboard), followed by the key whose control version is what you want. For <RETURN>, which is actually the same as CTRL-M, you therefore use ||M. Thus you could program key f0 to save you even having to press <RETURN> when renumbering, by

```
*KEY 0 RENUMBER ||M <RETURN>
```

Other control codes could be printed the same way. For instance, to switch into paged mode, LIST, and switch paged mode off automatically so that you do not accidentally forget, you could issue

```
*KEY 1 ||N LIST ||M ||O <RETURN>
```

Other commands frequently programmed into the function keys include RUN, LIST, OLD and mode changes. Although LIST, for example, has a two character abbreviation, programming a function key still reduces three keystrokes to one, counting the <RETURN> as well.

It is also possible to program the BREAK key to act as a function key (key number 10, which is logical, lying next to f9 as it does), although this does not alter its normal BREAK function which is carried out before issuing the programmed responses. It is quite common to use this feature to prevent users breaking out of a program, by programming the BREAK key to rerun the program with the following definition

```
*KEY 10 OLD ||M RUN ||M
```

Finally, if you need so many, the five cursor keys can be disabled and treated instead as yet more function keys, numbered 11 to 15, by the command

```
*FX 4,2
```

THE *TV COMMAND

The *TV command has two parameters and two purposes. The first use is to move the display up or down the screen. The default is *TV 0, but you can move the display up n lines by

```
*TV n
```

(there is a second parameter but this can be omitted unless you also want to switch off the interlace), or down n lines by

```
*TV 256-n
```

The command would be most useful to move the picture up if the current line of input is off the bottom of the screen. Note that you will not see the effect until after a mode change. Also, startling effects will be obtained if too large a value of n is used.

The second parameter to *TV determines whether the interlace is turned on or not. Interlace is the effect whereby a TV or monitor picture is built up of two interlaced halves containing alternate lines of the 625 line display. Although necessary on a TV set to display the complete picture, when displaying computer output it can give an unpleasant slight flickering effect. You can turn it off by the command

```
*TV 0,1
```

The two halves of the display will then be superimposed, giving a rock-steady picture. Note that the effect again only comes into operation after the next mode change (which could of course be into the same mode as before), and that interlace cannot be turned off in this way in Mode 7 (but see Section 6.4).

SOUND

Like graphics, the sound facilities on the BBC computer are very versatile, but the complex uses of sound, particularly the ENVELOPE command, are beyond the scope of this book.

The SOUND command by itself can be used to produce pure tones of any frequency, volume and length, and to mix up to three tones into chords. The command has the from

```
SOUND <channel>,<amplitude>,<pitch>,<duration>
```

We can briefly consider each of the parameters to SOUND in turn.

1. The first parameter, <channel>, can take four possible values, 0 to 3, but channel 0 is a special 'noise' channel. If a series of SOUND commands for different channels are issued in succession, this will produce the effect of chords.
2. The second parameter, <amplitude>, must be a number between 0 (zero amplitude) and −15 (maximum loudness).
3. The third parameter, <pitch>, determines the pitch, or frequency, of the tone. It is a number from 0 to 255, each step corresponding to a change in pitch of a quarter of a semitone. Since there are 12 semitones to an octave, an increase in the pitch number of 48 gives a one octave change in pitch. Middle C has a pitch value of 53, so C in the octave below is 5, and in the octave above is 101.
4. The final parameter, <duration>, is a number which can take a value from 1 to 255, and is in units of twentieths of a second.

Try the following sequence of SOUND commands

```
FOR J=1 TO 8: SOUND 1,-15,57+4*J,6: NEXT J
```

VDU COMMANDS

VDU commands are so called because they all govern display (on the VDU) in some way or other. Many are related to graphics and are dealt with in Chapter 6, but others are of more general application, and these are described below. All the VDU commands are summarized in Appendix C.

VDU 1 can be used to send a character to the printer but not to the screen. This is useful where a special code needs to be sent to a printer to switch on underlining or perform other functions of a similar type.

VDU 2 and VDU 3 are used before sending output to a printer and to terminate the output respectively. In immediate mode, the control-codes <CTRL-B> and <CTRL-C> are more convenient.

VDU 14 (CTRL-N) is very useful when listing programs or trying to examine large amounts of program output on the screen. About twenty lines of output at a time are displayed, then the computer pauses until the SHIFT key is pressed (no other key will do). Note that it remains in effect even when a program is subsequently running, and it can cause the computer to appear to 'hang', without reason. VDU 15 (CTRL-O) cancels the effect of VDU 14, but does not come into effect until the current listing or program run is ended.

VDU 21 (CTRL-U) has two different effects. If issued from the keyboard, it deletes the whole of the current line. From within a program, it has the effect of stopping all subsequent output (both text and graphics) from reaching the screen. This effect is cancelled by VDU 6.

Appendix B

SELECTED *FX COMMANDS

FX is short for 'effects', and is the command used for a range of special facilities or effects on the BBC microcomputer. The *FX commands can also be invoked by a call to OSBYTE, address $FFF4, with the *FX number in the accumulator (or A%). Certain calls return 'answers' in the X and Y index registers, and OSBYTE calls are the only way to use these particular effects. A very full list of the *FX calls is given in the *Advanced User Guide* by Bray, Dickens and Holmes, published by the Cambridge Microcomputer Centre. Below is given a selection of the *FX calls which are most useful to the BASIC programmer.

***FX 0**

Reports the version number of the operating system. (At the time of publication, the current operating system is 1.2, with earlier versions being 0.1 and (briefly) 1.0.)

***FX2,n**

Selects the source of input, which can be either the keyboard or the RS423 port.

n=0 selects the keyboard (default)
n=1 selects the RS423 port
n=2 selects both keyboard and RS423 port

***FX3,n**

Selects which output streams receive future output. The first three bits of n control: the RS423 output (bit 0, set to enable output), the screen (bit 1, clear to enable output) and the printer port (bit 2, clear to enable output). Thus *FX3,1 enables all three streams, *FX3,6 disables all streams. The default is n=0 (screen and printer enabled). The printer must still be selected by VDU 2, (or CTRL-B) but setting bit 3 will both enable *and* select the printer regardless of whether a VDU 2 has been issued.

***FX4,n**

Controls the action of the cursor control keys and the COPY key. n=0 gives normal editing (default), n=1 makes the keys act as normal keys with ASCII codes 135 to 139, while n=2 makes them act as extra programmable function keys, 11 to 15 (see Section 11.2.2 for further details).

***FX5,n**

Selects the printer type. n=1 selects the parallel printer port (default), n=2 selects the RS423 port, n=4 selects the Econet output. See Appendix K also.

***FX6,n**

Selects the character (ASCII code n) to be suppressed for printer output. See Appendix K for the use of this call.

***FX7,n and *FX8,n**

These calls set the baud rate for the RS 423 port for input (*FX7) and output (*FX8) according to the following table

n=1	75 baud
n=2	150 baud
n=3	300 baud
n=4	1200 baud
n=5	2400 baud
n=6	4800 baud
n=7	9600 baud
n=8	19200 baud

***FX9,n and *FX10,n**

Set the duration of the first (*FX9) and second (*FX10) flashing colours. n is the duration in fiftieths of a second (default n=25).

***FX11,n**

Sets the delay time before keys start to auto-repeat. n is the delay in centiseconds, n=0 disables auto-repeat (default n=32).

***FX12,n**

Sets the auto-repeat period to n centiseconds. A small value of n thus gives rapid repetition. n=0 restores the default value (8 centiseconds) and also the default for the auto-repeat delay.

***FX15,n and *FX21,n**

These calls flush, or empty, various internal buffers. *FX15,0 flushes all buffers, and *FX15,1 flushes the currently selected input buffer (see *FX2),

which will usually be the keyboard buffer. *FX21 flushes specified buffers, as follows

n=0	keyboard buffer (will usually be equivalent to *FX15,1)
n=1	RS423 input buffer
n=2	RS423 output buffer
n=3	printer output buffer
n=4–7	flush SOUND channels 0 to 3 respectively
n=8	speech synthesis buffer

***FX16,n**

Enables n of the ADC channels. (n=0 to 4. n=0 disables all channels.)

***FX17,n**

Forces the start of a conversion on ADC channel n.

***FX18**

Cancels any strings stored in the user-defined function keys.

***FX20**

Used to 'explode' or expand the number of user-defined characters. See Section 6.4.3.

***FX119**

Closes any files open for *SPOOL output or *EXEC input.

OSBYTE call 121 (&79)

This *FX must be made as an OSBYTE call, because a value is returned in the X register (see Section 10.8.4). If called with X=0, it scans the keyboard and returns the value of any key pressed, or 255 (&FF) if no key is pressed. Unlike INKEY(−X) it can be used not just to detect a particular key, but whether *any* key is pressed, and which it is. The number of the key pressed, which is returned in the X register, is related to the INKEY number, being ABS(INKEY number)−1. For example, SHIFT is 0, SHIFT LOCK is 80, → is 121.

OSBYTE call 131 (&83)

This OSBYTE call returns the value of the OSHWM (the OSHWM is the default value of PAGE – see Section 10.3) in the X and Y registers (low byte in X, high byte in Y).

OSBYTE call 132 (&84)

Returns the address of the bottom of the section of RAM being used for screen display (default value of HIMEM).

OSBYTE call 135 (&87)

Returns in the X register the character value of the character at the text cursor position, and in the Y register the graphics MODE number.

***FX138,m,n**

Inserts character with ASCII code n into the input buffer specified by m (m=0 for the keyboard buffer, and m=1 for the RS423 input buffer). See Section 10.7.2 for the way in which *FX138 can be used.

***FX200,n**

This affects the action of <ESCAPE> and <BREAK>. If bit 0 is set to 1 it will disable <ESCAPE>, and if bit 1 is set to 1 it will clear memory on <BREAK>. Thus *FX200,3 will prevent users from breaking out of a program with <ESCAPE>, and stop them getting back to the program with OLD after a <BREAK>.

***FX201,n**

*FX201,1 will disable the keyboard, *FX201,0 restores normal keyboard action.

***FX202,n**

As an OSBYTE call, this allows you to read (from the X register) the status of the keyboard locks etc, according to the following list, but is probably most useful as a means of controlling the SHIFT LOCK and CAPS LOCK from within a program, to force user input into the case desired for example. Bit 4 controls CAPS LOCK (cleared to 0 to set the lock) and bit 5 controls SHIFT LOCK (again 0 sets the lock). Thus *FX202,32 will switch on the CAPS LOCK. The full effect of all bits is

bit 3 returns 1 if SHIFT pressed
bit 4 1 for no CAPS LOCK
bit 5 1 for no SHIFT LOCK
bit 6 returns 1 if CTRL pressed
bit 7 1 to enable SHIFT

SHIFT enable allows the SHIFT key to work in reverse when a lock is in operation, so that you can type in lower case without disengaging CAPS LOCK or SHIFT LOCK.

***FX212,n**

Sets the amplitude for the BELL sound. The value of n should be set to 8× (<amplitude value>+31) where <amplitude value> is the usual number 0 to −15 as used in the SOUND command. The default value for n is 144.

***FX213,n**

Sets the pitch of the BELL sound. n is the normal pitch parameter of the SOUND command (default value 101).

***FX214,n**

Sets the duration of the BELL. n is the usual duration parameter of the SOUND command (default value 7).

***FX219,n**

Sets the ASCII code returned by the TAB key to be n. The default value is 9 (move cursor right) but it can be set to any value, including any control-code or soft key code. Thus *FX219,12 would turn the TAB key into a screen clear key.

***FX225–228,n**

These calls control the effect of the function keys in various combinations, as follows

`*FX225,n`	function keys alone (default n=1)
`*FX226,n`	SHIFT+function keys (default n=128)
`*FX227,n`	CTRL+function keys (default n=144)
`*FX228,n`	CTRL+SHIFT+function keys (default n=0)
n=0	makes the key combinations have no effect
n=1	makes them act as function keys
n>1	n acts as a base number, such that the key produces the ASCII code n+function key number

The default values for *FX226 and *FX227 are chosen to enable teletext colour and graphics codes to be produced readily.

***FX229,n**

n=1 causes the ESCAPE key to generate its ASCII code (27) instead of the normal ESCAPE action
n=0 restores the normal ESCAPE action

Appendix C

SUMMARY OF THE VDU COMMANDS

In the following list, items marked with an asterisk (*) are 'standard' effects of those particular control-codes, such as the <RETURN> character, CTRL-M. Items marked with a plus sign (+) are equivalents of BASIC commands.

VDU number	*Control character*	*Extra bytes required*	*Action*
0	@	0	None
1	A	1	Next character not sent to the screen (used to send output to the printer only)
2	B	0	Future output sent to the printer
3	C	0	Stop output to the printer
4	D	0	Cancel the effect of VDU 5
5	E	0	Future text written at graphics cursor position
6	F	0	Cancel the effect of VDU 21
*7	G	0	Create a short beep
*8	H	0	Backspace (move cursor left one space)
*9	I	0	Forwardspace (move cursor right one space)
*10	J	0	Linefeed (move cursor down one line)
*11	K	0	Reverse linefeed (move cursor up one line)
+*12	L	0	Clear text screen (new page, CLS)
*13	M	0	<RETURN> (carriage return)
14	N	0	Switch on page mode for output
15	O	0	Switch off page mode
+16	P	0	Clear graphics screen (CLG)
+17	Q	1	COLOUR command

VDU number	*Control character*	*Extra bytes required*	*Action*
+18	R	2	GCOL command
19	S	5	Reassign logical colour
20	T	0	Cancel all VDU 19 assignments
21	U	0	Immediate mode: cancel line Program mode: switch off future output
22	V	1	Changes display mode but does *not* reset HIMEM
23	W	9	Specify user-defined character
24	X	8	Set graphics window
+25	Y	5	PLOT command
26	Z	0	Cancel VDU 24 and VDU 28 and home text and graphics cursors
*27	[	0	ESCAPE character. When issued by a program, does nothing
28	\	4	Set text window
29	]	4	Set a new graphics origin
30	^	0	Home text cursor to top left of screen
+31	_	2	TAB(X,Y)
+127		0	DELETE character

Appendix D

BUGS AND UNEXPECTED EFFECTS IN BBC BASIC

Inevitably there are some bugs in the BBC microcomputer software. A comprehensive list is published by the MEP Software Unit. The MEP list includes bugs in the assembler, network filing system and quirks as well as bugs. In this appendix the most serious bugs in those parts of the system covered by this book are reported.

1 Renumbering on BREAK and OLD

When BREAK is pressed, the first byte of the program in memory is set to &FF to clear the program. This byte normally represents the high byte of the first line number. When the OLD command is issued, it simply sets the first byte of the program to zero to remove the end of program marker. If the line number was originally greater than 255, it will be reduced to the low byte of the number – e.g. 300 will become 44, 1000 will become 232.

2 DIM statement

The BASIC interpreter should trap DIM statements which require more memory than is available, and report a 'Bad DIM' error. Occasionally the interpreter misses the error; this occurs when at least the lower six bits of every dimension are set – e.g. DIM A(63,127,255). Other dimensions such as DIM A(64,126,255) will be trapped correctly. The result can be disastrous, because the interpreter clears all the memory set aside for the array, and may even overwrite crucial areas of memory used by the system. The only solution then will be to switch off the computer.

3 DATA statement

A DATA statement must be the first statement on a line, or it will be ignored. This is not perhaps a bug, but certainly a limitation of which you should be aware.

More seriously, if the special BBC feature of adding a line number to RESTORE is used to set the DATA pointer to that line instead of back to the very start of the program, data will be read from that line even if it is not a

DATA statement. In this case, data is assumed to start after the first comma on the line.

4 RENUMBER command

If, on RENUMBER, line numbers would exceed 32767, they are treated MOD 32768, that is they 'wrap round' instead of generating an error.

5 Undefined variables

A useful safety feature of BBC BASIC is that variables must be assigned a value before they can be used. There is an exception to this when the variable appears on both sides of an assignment statement. The initial value of the variable is taken as zero, so the statement

```
J=J+10
```

will result in J taking the value 10 if it has not previously been assigned a value.

6 INKEY statement

If INKEY(X) is used with a value of X greater than 255, then the first time that the statement is executed, the effective value of X is reduced by 256. The solution is to issue a previous INKEY with an argument of 1.

7 CLEAR statement

Users should be aware that the CLEAR statement does not merely clear variables, it also empties the BASIC stack and clears the data pointer and LOMEM. The BASIC stack contains addresses of the start of loops and return addresses of subroutines, procedures and functions, so CLEAR should never be issued from within a FOR or REPEAT loop or sub-program.

8 Variable names

Variable names can include underline symbols and pound signs as well as letters and numbers. The former is particularly useful to give an impression of a space, as in New_X.

9 Double-height characters

When using double-height characters in Mode 7, the line immediately below a line containing the double-height character 141 becomes a special line. Characters following another character 141 generate lower halves of the characters, thus enabling the double-height characters to be completed, but 'normal' characters, before the 141 or after a character 140, do not show up. This is not really a bug, but can be a restriction in certain circumstances. A consequent effect is that, when the upper of a pair of double height character lines scrolls off the top of the screen, the second line acts as an upper half and suppresses normal characters on the next line.

10 Incorrect catalogue

When discs are in use, the latest catalogue information is stored in the computer on pages 14 and 15 (&E and &F). Under certain circumstances, particularly if the disc drive has not stopped spinning since the last disc access, the DFS assumes that it still knows what the disc contains and does not read the catalogue again. If you have swapped discs in the meantime, the catalogue information would be incorrect, and any writing process using the incorrect catalogue could be a disaster.

Try loading a program, and before the disc drive has stopped spinning, open the drive door and remove the disc. The drive will keep running, and while it is empty type *CAT. The old catalogue list will appear even though the drive is empty.

The key point to remember is always to let the drive stop spinning before issuing any command that writes to a disc.

11 Limitations on *SPOOL

Text produced by the DFS cannot be written into an ASCII file.

The following bugs occur in BASIC I, but have been corrected in BASIC II.

12 ABS function

In BASIC I, ABS may not be preceded by the unary minus operator (i.e. a minus sign to invert the sign of the function, rather than to subtract it from another value). Thus

```
PRINT -ABS(-1)
```

is not allowed, although

```
PRINT 3-ABS(-1)
```

is permissible.

13 COUNT function

In BASIC I, the value returned by COUNT is not zeroed after a mode change, so that after a mode change in the middle of a line of output COUNT would include the old line in its total.

14 ON...ELSE

The structure of combining ELSE with ON...GOTO and ON...GOSUB is not allowed within procedures and functions in BASIC I.

15 EVAL function

In BASIC I, the EVAL function will not correctly evaluate pseudo-variables. Instead it treats them as ordinary variables, and reports 'no such variable'.

16 INSTR command

In BASIC I, this command contains a serious bug whereby the stack is corrupted if the search string is larger than the original string. This bug is explained more fully in Section 4.4.

17 ON ERROR trap

Certain line numbers cannot be used with ON ERROR GOTO in BASIC I. Line numbers such as 9999 cause the machine to hang up completely if an error occurs.

Appendix E

DIFFERENCES BETWEEN BASIC I AND BASIC II

The second release of BASIC on the BBC microcomputer has a number of improvements over the original version, and these are listed below. In addition, a number of the bugs have been corrected. These bugs are described in Appendix D, and are simply listed below in brief.

The first release of BASIC is now commonly referred to as BASIC I, and the revision is called BASIC II. The easiest way to determine which version of BASIC you have is to press <BREAK>, and then type the command REPORT. The computer replies with a copyright message, which includes the dates 1981 and 1982 for BASIC I and BASIC II respectively.

The following bugs have been corrected in BASIC II.

- ABS can now be used with a preceding minus sign.
- COUNT is now properly reset on a mode change.
- ELSE may now be used with ON...GOTO and ON...GOSUB within procedures and functions.
- EVAL will now handle pseudo-variables.
- INSTR can now handle a search string longer than the source string.
- ON ERROR will now handle all line numbers correctly.

The following improvements have been made in BASIC II

INPUT

This will now accept the semicolon as a separator for all items, as well as the comma. This brings it into line with other implementations of BASIC, where the command has the form

```
INPUT "leading string";A,B,C
```

OPENUP

The command OPENUP is introduced for opening data files for reading *and* writing (i.e. UPdating), and OPENIN is reserved for reading only. This feature is fully discussed in Section 9.3.1.

OSCLI

The command OSCLI followed by a string or string variable passes the string to the command line interpreter. In BASIC I it was necessary to load the string into memory, set a pointer to the string via X% and Y% and CALL &FFF7. Use of the CLI is discussed in detail in Section 10.7.1.

LN, LOG, SIN, COS

These functions have all been increased in accuracy. The effect will be most noticeable near the limits of range of the functions.

PRINT (and STR$)

Numbers are printed with slightly greater precision.

EQUB, EQUW, EQUD, EQUS

Four new assembler pseudo-operations are provided, to transfer values to bytes within machine code. The operations respectively define a single byte, word (double-byte), double-word (four bytes) and string.

The instructions are used in statements such as

```
.values EQUB &0D
 EQUW &1234
 EQUD &01020304
 EQUS "END OF MESSAGE"
```

which store the specified values in memory starting at the address specified by the label 'values'.

Appendix F

THE RESIDENT VARIABLE @%

The detailed features of the layout of numbers output by the PRINT command are governed by the resident integer variable @%. Like all integer variables it is a four byte number, and each byte serves a separate function. It is most conveniently specified in hexadecimal (that is, with a leading &) since then each pair of digits represents one byte. If the four bytes have values B4, B3, B2 and B1, then for the number

```
@%=&12345678
```

we would have B4=&12, B3=&34, B2=&56 and B1=&78.

Alternatively, if B1 and B4 are treated as variables with separate values, then @% can be denoted by

```
@%=B1+&100*(B2+&100*(B3+&100*B4))
```

or

```
@%=B1+256*(B2+256*(B3+256*B4))
```

which will be the best way to set up @% if you want to vary it from time to time.

B4

B4 has a rather marginal effect – it determines whether the format of a string created by STR$ is controlled by @%. If B4=01, @% is used in determining the format, if B4=00, @% is ignored. The default value for B4 is zero and this is why, as pointed out in Section 4.4.2, STR$ may not always produce quite what you would expect.

B3

B3 selects from 3 types of format as follows

B3=00	General format (or G format)
B3=01	Exponent format (or E format)
B3=02	Fixed format (or F format)

G format is the default state, in which a number is printed as a straightforward fixed point number (for example 123.45) if it is neither too large for the specified range, nor less than 0.1. Outside these limits it is printed in exponent form or scientific notation as a floating point number, with a mantissa (a fixed point number between 1 and 10) and an exponent or power of 10. Thus 0.0123 would be written as 1.23E−2 and 12 milliard would be written as 1.2E10.

In E format, numbers are always written in exponent form.

F format is the most interesting. All numbers are printed with a fixed number of decimal places (determined by B2) unless they exceed the value which can be stored internally as an exact number (more than nine figures) in which case they revert to exponent form. Numbers which are too small, however, are printed as zero. F format is designed for, and is ideal for, tabulated columns of numbers, so that, combined with right justification, decimal points will be aligned.

B2

B2 is the other factor affecting the format of printed numbers. It plays different roles according to the value of B3.

In G format, it specifies the maximum number of significant figures to be printed, and hence also the largest number which can be printed in fixed point, before reverting to exponent form. The default value is 9, which is one figure less than the machine accuracy, so that rounding errors are normally suppressed.

In exponent form numbers, B2 again determines the maximum number of significant figures of the mantissa. Including a possible leading minus sign, the decimal point, the E, up to two digits for the power of 10 and another possible minus sign for the power, exponent form numbers in G format occupy up to B2+6 characters in all. One difference occurs for E format numbers, in that the power occupies three columns, including trailing spaces, and B2 figures are always printed in the mantissa, so numbers always occupy B2+5 or B2+6 columns, depending on whether there is a minus sign.

In F format, B2 plays a quite different role, specifying the number of digits to follow the decimal point. The maximum number of significant figures is always set to nine, being governed by the machine accuracy, so that numbers do not revert to exponent form below 10^9. Even integers are printed with a decimal point and B2 trailing zeros. Note, however, that if the number of figures printed would exceed nine, the display reverts to G format. For example, with B2=4, 123456 would be printed as an integer, whereas with B2=3 it would be printed as 123456.000.

The allowed range of B2 is as follows

G format 1 to 9
E format 1 to 9
F format 0 to 9

For numbers outside these limits, B2 reverts to the default of 9.

As an example of the use of B2, Table F.1 shows how the number 12.3456 would be printed in each of the formats, for values of B2 of 3, 6 and 9.

Table F.1 Effect of B2 on the format of numbers.

	G format	*E format*	*F format*
B2=9	12.3456	1.23456000E1	12.3456
B2=6	12.3456	1.23456E1	12.345600
B2=3	12.3	1.23E1	12.346

Note that with B2=9, the F format display has reverted to G format.

B1

As we saw earlier, B1 sets the print field width. This is not the maximum number of characters that will be printed, but is simply the width between each 'tab position' as determined by commas in PRINT statements. It does, however, limit the number of characters that will be printed right justified, and hence with the decimal points aligned. Moreover, right justification will only occur if items are separated by commas, so in principle this form of layout would have to be in columns of equal width. In practice, the TAB(X,Y) form overrides the left margin origin from which B1 measures, so it could be used to set out columns of figures of different widths (but not less than B1). The range of B1 is any value from 0 to 255. However, the practical upper limit is probably more like 20.

Example F.1

The following program generates numbers whose magnitudes and number of figures, as well as values, are random, and prints them in successive columns of width 10, 13, 17, 12, 20 in Mode 0, with respectively 0, 3, 4, 2, 5 decimal places, properly aligned.

```
10 MODE 0
20 DIM A(5)
30 CLS
40 FOR ROW=1 TO 20
50 FOR COL=1 TO 5
60 REPEAT
70 A(COL)=FN_rannum
80 UNTIL ABS(A(COL))<1E8
```

```
 90 NEXT COL
100 @%=&2000A: PRINT TAB(0,ROW) A(1)
110 @%=&2030D: PRINT TAB(10,ROW) A(2)
120 @%=&20411: PRINT TAB(23,ROW) A(3)
130 @%=&2020C: PRINT TAB(40,ROW) A(4)
140 @%=&20514: PRINT TAB(52,ROW) A(5)
150 NEXT ROW
160 END
1000 DEF FN_rannum
1010 LOCAL DECPLACES, NUM
1020 DECPLACES=10(RND(6)-1)
1030 NUM=INT((RND(1)-0.5)*10RND(9))
1040 =NUM/DECPLACES
```

Note that the PRINT statements must *not* end with semicolons. The TAB(X,Y) form forces all the output for one row to appear on the same line. This cannot, of course, be used in conjunction with a printer.

Note also that the formatting still goes wrong occasionally for very large numbers, especially in columns 3 and 5 where several decimal points are required. This can be prevented by restricting the size of the numbers to less than 10^6 (alter the last RND on line 1030 to RND(5)). It is of course a very demanding test to format numbers of such varying type, but it does serve to demonstrate that although the @% variable provides great flexibility, it is difficult to use.

One final property of B1 may sometimes be useful. Provided it is set to at least 2, numbers separated by commas will be printed with at least one space between them. Thus if a series of variables is to be printed where alignment of columns is immaterial, but where the normal large gaps are unnecessary, setting B1 to 2 will be much more convenient than separating each variable by " ".

Appendix G

PSEUDO-VARIABLES IN BBC BASIC

The following are BBC BASIC pseudo-variables. They always have a value, and may be altered by the system as well as by the user, but in other respects they can be handled just like ordinary variables.

HIMEM	Pointer to top of available RAM
LOMEM	Pointer to the bottom of variable storage space
PAGE	Pointer to the start of program storage (must be a multiple of 256)
PTR#	Pointer to position in a data file
TIME	An internal timer reading in centiseconds

The following are strictly functions, but could be thought of as read-only pseudo-variables.

COUNT	Counts the number of characters printed since the last <RETURN>
EOF#	Boolean variable which becomes true when the end of a data file is reached
ERL	Line number where the last error occurred
ERR	Error number of the last error (see Appendix J)
EXT#	Current size of data file
FALSE	The Boolean value 'false' (returns a numerical value of 0)
PI	The mathematical quantity π, with value 3.14159265
POS	Horizontal position of the cursor (left hand side=0)
TOP	Pointer to the top of a program
TRUE	The Boolean value 'true' (returns a numerical value of -1)
VPOS	Vertical position of the cursor (top=0)

Appendix H

ASCII CODES

Character	*Decimal*	*Hex*
CTRL-@	0	00
CTRL-A	1	01
CTRL-B	2	02
CTRL-C	3	03
CTRL-D	4	04
CTRL-E	5	05
CTRL-F	6	06
CTRL-G	7	07
CTRL-H	8	08
CTRL-I or TAB	9	09
CTRL-J	10	0A
CTRL-K	11	0B
CTRL-L	12	0C
CTRL-M or RETURN	13	0D
CTRL-N	14	0E
CTRL-O	15	0F
CTRL-P	16	10
CTRL-Q	17	11
CTRL-R	18	12
CTRL-S	19	13
CTRL-T	20	14
CTRL-U	21	15
CTRL-V	22	16
CTRL-W	23	17
CTRL-X	24	18
CTRL-Y	25	19
CTRL-Z	26	1A
CTRL-[or ESC	27	1B
CTRL-\	28	1C
CTRL-]	29	1D
CTRL-^	30	1E
CTRL-_	31	1F

Character	*Decimal*	*Hex*
SPACE	32	20
!	33	21
"	34	22
#	35	23
$	36	24
%	37	25
&	38	26
'	39	27
(	40	28
)	41	29
*	42	2A
+	43	2B
,	44	2C
-	45	2D
.	46	2E
/	47	2F
0	48	30
1	49	31
2	50	32
3	51	33
4	52	34
5	53	35
6	54	36
7	55	37
8	56	38
9	57	39
:	58	3A
;	59	3B
<	60	3C
=	61	3D
>	62	3E
?	63	3F

Character	*Decimal*	*Hex*
@	64	40
A	65	41
B	66	42
C	67	43
D	68	44
E	69	45
F	70	46
G	71	47
H	72	48
I	73	49
J	74	4A
K	75	4B
L	76	4C
M	77	4D
N	78	4E
O	79	4F
P	80	50
Q	81	51
R	82	52
S	83	53
T	84	54
U	85	55
V	86	56
W	87	57
X	88	58
Y	89	59
Z	90	5A
[	91	5B
\	92	5C
]	93	5D
^	94	5E
_	95	5F

Character	*Decimal*	*Hex*
£	96	60
a	97	61
b	98	62
c	99	63
d	100	64
e	101	65
f	102	66
g	103	67
h	104	68
i	105	69
j	106	6A
k	107	6B
l	108	6C
m	109	6D
n	110	6E
o	111	6F
p	112	70
q	113	71
r	114	72
s	115	73
t	116	74
u	117	75
v	118	76
w	119	77
x	120	78
y	121	79
z	122	7A
{	123	7B
‖	124	7C
}	125	7D
~	126	7E
DELETE	127	7F

Appendix I

INKEY CODES

When the keyboard is examined directly by using INKEY with a negative argument, the code for the key has no relationship to the ASCII code of the character, but is an internal code related to the structure of the keyboard. A closely related table of codes is used by OSBYTE call 121.

INKEY code	*OSBYTE 121 code*	*symbol*
-1	0	<SHIFT>
-2	1	<CTRL>
-17	16	Q
-18	17	3
-19	18	4
-20	19	5
-21	20	f4
-22	21	8
-23	22	f7
-24	23	-
-25	24	^
-26	25	←
-33	32	f0
-34	33	W
-35	34	E
-36	35	T
-37	36	7
-38	37	9
-39	38	I
-40	39	O
-41	40	
-42	41	|
-49	48	1
-50	49	2
-51	50	D
-52	51	R
-53	52	6
-54	53	U
-55	54	O
-56	55	P
-57	56	[
-58	57	↑
-65	64	<CAPS LOCK>
-66	65	A
-67	66	X
-68	67	F
-69	68	Y
-70	69	J
-71	70	K
-72	71	@
-73	72	:
-74	73	<RETURN>
-81	80	<SHIFT LOCK>
-82	81	S
-83	82	C
-84	83	G
-85	84	H
-86	85	N
-87	86	L
-88	87	;
-89	88	]
-90	89	<DELETE>
-97	96	<TAB>
-98	97	Z
-99	98	<SPACE>
-100	99	V
-101	100	B
-102	101	M

INKEY code	*OSBYTE 121 code*	*symbol*
−103	102	,
−104	103	.
−105	104	/
−106	105	<COPY>
−113	112	<ESCAPE>
−114	113	f1
−115	114	f2

INKEY code	*OSBYTE 121 code*	*symbol*
−116	115	f3
−117	116	f5
−118	117	f6
−119	118	f8
−120	119	f9
−121	120	\
−122	121	←

Appendix J

ERROR CODES

Code	*Error*
0	No room (cannot be trapped)
0	STOP (BASIC II only)
1	Out of range (assembly language error)
2	Byte (assembly language error)
3	Index (assembly language error)
4	Mistake
5	Missing ,
6	Type mismatch
7	No FN
8	$ range (Trying to address locations less than 256 (&100) with the $ indirection operator)
9	Missing "
10	Bad DIM
11	DIM space
12	Not LOCAL
13	No PROC
14	Array
15	Subscript
16	Syntax error
17	Escape
18	Division by zero
19	String too long
20	Too big
21	-ve root
22	Log range
23	Accuracy lost
24	Exp range
25	Bad MODE
26	No such variable
27	Missing)
28	Bad hex
29	No such FN/PROC
30	Bad call

Code	*Error*
31	`Arguments`
32	`No FOR`
33	`Can't match FOR`
34	`FOR variable`
35	`Too many FORs` (only 10 FORs may be nested)
36	`No TO`
37	`Too many GOSUBs` (only 26 nested subroutines allowed)
38	`No GOSUB`
39	`ON syntax`
40	`ON range`
41	`No such line`
42	`Out of DATA`
43	`No REPEAT`
44	`Too many REPEATs` (only 20 REPEATs may be nested)
45	`Missing #` (BASIC II only)
216	`Data?` (cassette filing system error)
217	`Header?` (cassette filing system error)
218	`Block?` (cassette filing system error)
219	`File?` (cassette filing system error)
220	`Syntax` (filing system error)
222	`Channel` (filing system error)
223	`Eof` (filing system error)
251	`Bad key`
253	`Bad string`
254	`Bad command`

The following errors have no error code

`Bad program` (cannot be trapped)
`Failed at <line number>` (error during RENUMBERing)
`LINE space` (no room in memory to insert line)
`Silly` (occurs with AUTO if a step size greater than 255 is specified)

Appendix K

PRINTERS

Printers are categorized according to the method they use to produce print and how they interface to the computer. The BBC Model B microcomputer can be used with most present-day printers and a brief discussion of the more popular types and how they are used is given in this appendix.

DOT-MATRIX PRINTERS

The most popular type of printer is the dot-matrix printer which, as the name implies, has a print head with a matrix of pins, typically 5×7 although some are available with many more pins, giving a much better quality print. Producing characters as an array of dots provides flexibility in the types of characters produced, and a common facility is that of several, switch selectable, character sets, for use in different countries – UK, USA, Germany and Sweden for example. Another advantage is that some, but not all, dot-matrix printers can be used in graphics mode. This means that in conjunction with a suitable program often called a screen dump, they can be used to produce a hard copy of the contents of the screen when the computer is in graphics mode. Dot-matrix printers are relatively inexpensive.

DAISY-WHEEL PRINTERS

For high quality production of text, the more expensive daisy-wheel type of printer is superior. The characters are, of course, fixed by the daisy-wheel but the wheels can be interchanged to produce different size and type of characters, including Greek letters and mathematical symbols. Certain more expensive models are available with twin daisy wheels which are software selectable. Daisy-wheel printers cannot easily be used to produce a screen dump.

INTERFACING

A printer can be either a serial or parallel type, depending upon the way it interfaces to the computer. A serial type can normally be driven with just

three wires and because of the low-cost and compactness of the cable is suitable where the printer is to be stationed at some distance from the computer or is to be switched to serve a number of machines. Where the printer is to be used adjacent to the computer, then serial or parallel types are equally suitable although the latter may have a small advantage in terms of cost.

The BBC computer provides a parallel printer port, often referred to as a Centronics compatible port, implying that it satisfies prescribed standards for interconnections and protocol. It emerges via a 26-way socket accessible from the front underside of the computer case and is connected using a 26-way ribbon cable. For connection to a serial printer, the serial RS423 port is used. This is accessed from the rear via a 5-pin domino DIN socket.

USE OF PRINTERS

In order to send output from the computer to the printer it is necessary to give the computer a series of commands. On power-up the computer will automatically select the parallel port for printer output. If a serial printer is being used then the serial port can be selected by means of the command *FX5,2. The parallel port can be reselected by means of the command *FX5,1.

When printing, an additional option is to switch off the screen display. This is just one of the uses to which *FX3 can be put. If you would like to send output to the printer without it appearing on the screen, issue *FX3,3 before the output and *FX3,0 after output is completed. Note that *FX5,2 must *also* be issued if necessary. In principle, an alternative to both *FX commands is to simply issue *FX3,7, which disables the screen and enables the RS423 port simultaneously. However, this may not be advisable since it merely selects the RS423 socket as the output stream; it does not recognize it as a printer output stream so the *FX6 command, for instance, will be ineffective.

In the case of a serial printer the rate at which information is transmitted will have to be adjusted to match one which the printer is capable of receiving. Usually this would be the highest rate available, provided that the printer has 'handshaking' signal lines to stop the computer transmitting when it cannot receive any more information.

Information transmission rates are measured in baud with a typical value for printers being 300 baud.

Transmit baud rates can be changed with the following statements

```
*FX8,1      75 baud
*FX8,2     150 baud
*FX8,3     300 baud
*FX8,4    1200 baud
```

(Higher baud rates are possible, but are unlikely to be of use for printer output.)

Once the port has been selected and, in the case of the serial printer, the baud rate has been selected, the printer can be *enabled*. This means that the computer is instructed that all output that goes to the screen is also to be sent to the printer. It can be done directly from the keyboard by means

of CTRL-B. Similarly, the printer can be turned off by means of CTRL-C. To turn a printer on and off during execution of a program the statements VDU 2 and VDU 3 can be used respectively (or *FX 3,8 – see Appendix B).

In addition to the parallel and serial printer outputs, computers connected to an Econet network can use communal printer facilities provided by the network. The Econet output is selected by *FX5,4.

CHARACTER SUPPRESSION

Some printers automatically line feed when a carriage return is received. Therefore the effect of a carriage return and line feed being transmitted to this type of printer would be to produce a double line feed. This problem can be overcome by instructing the computer to suppress line feeds. The command *FX6,n will suppress the character with ASCII code n and the command *FX6,0 will cancel any previous suppressions. The ASCII code for line feed is 10 and therefore *FX6,10 will suppress line feeds. The default on the BBC computer is in fact to ignore line feeds on power-up. Therefore, if the printer does not automatically line feed, the command *FX6,0 must be used to enable line feeds.

If you do not need to suppress line feeds, you could use the *FX6 command to economize on paper during printer output from within a program by issuing *FX6,12, which will suppress any page feeds generated by CLS commands in the program.

The sequence of commands to list a program is therefore as follows

Parallel

```
*FX5,1              (necessary only if serial port previously selected)
*FX6,0              enable line feeds (if required)
CTRL-B or VDU 2     turn on
LIST.
CTRL-C or VDU 3     turn off
```

Serial

```
*FX5,2              select serial port
*FX8,n              n=1, 2, 3 or 4 according to baud rate
*FX6,0              enable line feeds (if required)
CTRL-B or VDU 2     turn on
LIST
CTRL-C or VDU 3     turn off
```

Econet

```
*FX5,4              select Econet
*FX6,0              enable line feeds (if required)
CTRL-B or VDU 2     turn on
LIST
CTRL-C or VDU 3     turn off
```

DETECTING THE PRESENCE OF A PRINTER

If output is directed to a printer when no printer is connected, the computer will hang once the output buffer is filled. A solution is to send the first N characters to the printer (N must be in the range 2<=N<=63), then check the free space in the printer buffer by

```
X=ADVAL(-4)
IF X=64-N THEN PRINT CHR$(3);"NO PRINTER CONNECTED": GOTO...
```

If the free space is not greater than 64−N, then nothing has been transmitted and preventive action can be taken.

Appendix L

THE ECONET FILING SYSTEM

A computer network is a system for linking together a number of computers to permit communication between users, and also to enable them to share resources such as common disc storage and printer. It acts as an alternative filing system for the computers attached to the network.

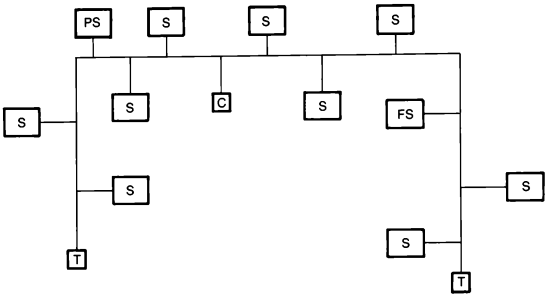
The Econet is a network designed for use with all Acorn computers. The block diagram in Figure L.1 shows a possible configuration for this network.

To connect computers to the Econet it is necessary to install an interface. This involves the use of a special ROM for software and the MC68B54 Advanced Data Link Controller which forms part of the necessary hardware. The ROM contains the software for a filing system called the Network Filing System (NFS) which is similar in some respects to the Disc Filing System (DFS) and the Cassette Filing System (CFS). The network filing system is selected by the command

```
*NET
```

In an Econet system up to 254 BBC microcomputers can be connected together. Each microcomputer connected to the network has a station identity which is a number between 1 and 254. This number is set by a switch inside the computer case. No two computers on a network can have the same station identity. The diagram of the network shows that, in addition to the user stations, there are usually two special purpose computers: the file server, which together with a disc drive provides storage for the network and a printer server, which together with a printer, is available for program listings and output of data. A network may have more than one file server or printer server.

The printer server can be used as a normal station except when it is accessed by another user on the network to print a file. The printer server needs an additional special ROM containing the printer software. The network computers assume that the file server has a default identity of 254 and that the printer server has the default address (station identity) of 235. The computer (or computers) that acts as a file server is dedicated to that purpose and runs a special program supplied on a floppy disc. Two versions of



Key

C clock
FS file server
PS printer server
S station
T terminator

Figure L.1 The arrangement of a network.

the file server software have been released and are in widespread use. These are referred to as level 1 and level 2.

Computers that have the Econet system installed will usually be arranged to default to the Econet filing system on switch-on and after a <BREAK>. It is possible to cause a <BREAK> or switch-on to select another filing system even though the Econet system is installed, as follows

<CTRL-D>+<BREAK> selects DFS (Disc Filing System)
<CTRL-T>+<BREAK> selects CFS (Cassette Filing System)
<CTRL-N>+<BREAK> selects NFS (Network Filing System)

Assuming that the file server has been started by the system manager, and the network is in operation, then each user station can access the Econet filing system.

After a switch-on with the network selected, a computer fitted with the Econet will respond thus

```
BBC Computer 32K
Econet station nnn
BASIC
>
```

where nnn is the station identity. The other filing systems can be selected in the usual way by *DISC or *TAPE without loss of program or data. Thus programs may be transferred between different filing systems.

LEVEL 1 ECONET

The range of facilities available to the network user, and the way in which they can be accessed by the Econet station user, depend upon which level of Econet the file server is running. A network can have file servers working with either level at the same time. The protocols for the transfer of data on the network are independent of the level of software. However the types of data that can be transferred on the network are limited by the level of Econet software.

On level 1 the user can select a file server by the command

```
*I AM nnn
```

where nnn is the station identity of the file server selected. As the default station number is 254 it is not necessary to select this after switch-on.

On level 1 the file server has a list of valid users identified by a single letter from A to Z, which is also the name of their directory on the file server. The station identity is paired with the user name/directory so that each station can only have access to one directory. The directories for different stations can all be different or some can have common directories. Thus a station cannot have more than one directory assigned to it, but a directory can have several stations assigned to it. The allocation of directories to stations is set up by the system manager whenever he starts the system up. This means that the users must normally use the same machine each time, in order to have access to their own files.

The level 1 file server software can be run on a Model B microcomputer with any type of disc drive and Econet interface. It does not require the 6502 second processor, which is needed by level 2 file servers. The level 1 file server is a BASIC program which uses the DFS filing system. This limits the total number of files available to the Econet system users to a maximum of about 120 on each file server and that only if a twin double-sided disc drive is being used.

In addition to the individual directories indicated by a letter, allocated to each network station, there is a library directory, &, which is available to all users on the network. The library cannot be written to while the system is running but since the file server discs are in normal disc filing system format, they can be written to whilst the file server is operating as an ordinary DFS machine. The network computers automatically select the & directory on

drive 0 as the library. The & directory can only be accessed on drive 0; it can not be accessed on any other drive. Thus the library must be on drive 0. The library directory & is intended for the system utility programs which are used by all users, and for demonstration programs for the whole class to use. One of the advantages of the level 1 Econet system is that the discs can be prepared on an ordinary DFS machine, unlike level 2.

Another advantage of the Econet system is that it can provide a printer server. This means that a whole group of users can access a common printer. An Econet network can, however, have a printer server without having a file server.

The range of commands available on the level 1 Econet system is restricted and these are listed in Table L.1. The absence of data file handling

Table L.1 Level 1 Econet system commands.

Command	*Argument*	*Explanation*
`SAVE`	`"XYZ3789"`	Save a BASIC program in file XYZ3789.
`LOAD`	`"XYZ3789"`	Load the BASIC program XYZ3789.
`*CAT`		Catalogue the files of a particular user.
`*DELETE`	`<fsp>`	Delete a specified file.
`*DRIVE`	`n`	Select the disc drive surface in use (n=0 to 3).
`*ACCESS`	`<fsp> L`	Lock (L) or unlock ( ) a file. Locked files cannot be deleted or overwritten.
`*I AM`	`nnn`	Select file server by stated identity.
`*SAVE <fsp> start finish` `*SAVE <fsp> start +length`		Save an area of memory on disc giving start address and finish address or length.
`*LOAD`	`<fsp>`	Load file into memory.
`*RUN`		Execute a machine code program.
`*INFO`	`<fsp>`	Displays information about a file.
`*<fsp>`		Execute a machine code program as a command.
`*/`	`<fsp>`	Allows <fsp> to take reserved names such as CAT so that you can write your own utilities.
`*PS`	`nnn`	Select station nnn as print server.
`*FX 5,4`		Special *FX call to select Econet printer. Use CTRL-B and CTRL-C in the usual way to start and finish printing.

commands is a severe limitation to the use of level 1. Another restriction is that, as the system is run by a file server without a second processor, the maximum length of file that can be transferred across the network is also limited. The latter restriction can be overcome by saving blocks of memory with *SAVE and using *LOAD to reassemble a file which would otherwise be too large.

The file server disc drives may be single or dual, double-sided or single-sided drives. If two single-sided drives are used, then only drives 0 and 1 are available, whereas if double-sided drives are used, then drives 2 and 3 can also be used. The system and library files are stored on drive 0 which means that there is limited space on this drive. It is therefore advisable for users to save programs to one of the other drives. You can change drive by use of the command

```
*DRIVE n
```

where n=0, 1, 2 or 3, and subsequent LOAD or SAVE commands will refer to the selected drive. Alternatively, you can SAVE or LOAD from a different drive without changing the selected drive by preceding the filename with :n. where n is the drive number.

The commands in Table L.1 form part of the Econet software and are always available to the network user. A second group of commands or utilities would normally be made available to the user at the discretion of the system manager and these are given in Table L.2. A utility is a separate program that can be used in the same way as a command. The utility programs are stored in the & directory and are *RUN by the *<filename> command which will search the selected library if a file of the given name is not in the user's directory. The system manager can choose which utilities are available to the users by the range of program files he provides.

Table L.2 Level 1 network utilities.

Command	*Argument*	*Explanation*
*PROT		Protect station from remote control.
*REMOTE	nnn	Take remote control of station nnn.
*VIEW	nnn	Take copy of VDU from station nnn.
*NOTIFY	nnn	Send message to nnn.
*UNPROT		Unprotect station.
*FS	nnn	Change file server to station nnn.
*ROFF		Remote off.

The Econet Level 1 system can generate a range of error conditions outside those met in a non-Econet environment. These are listed in Table L.3.

Table L.3 Additional Level 1 error messages.

Message	*Explanation*
`Directory full`	The number of files each user may store is restricted at the start-up of the system. You cannot exceed this limit.
`File not found`	The Econet system only looks on the selected drive for a requested file.
`Insufficient access`	The file server does not recognize you as a user station. The identity of the computer is not in the list given.
`Not listening`	Your command has not been accepted by the file server or print server.
`Too much data`	This arises when the file length limit is exceeded.
`Line jammed`	Information permanently present on the network.
`No clock`	The network clock signal is not reaching the station.
`No reply`	The file server is not responding to your command.

The last three error messages require the attention of whoever is in charge of the network. The procedure for starting the network is not discussed here since this will usually be the responsibility of a designated person and this text is intended for the general user and not the system manager.

LEVEL 2 ECONET

The level 1 Econet system provides a very rudimentary network capability. The level 2 system provides a full range of facilities so that individual station users have nearly the same range of facilities as an individual user with the DFS and a disc drive. In addition, the station user is able to work in an environment akin to that of a mainframe computer system in terms of access to the files of a wide range of other users.

The level 2 Econet system supports a hierarchical directory system and does not use the standard disc filing system (DFS), so several of its limitations are removed.

The root directory is the $ or system directory and SYST is a user who has access to this directory for all purposes. New directories and users can be created by SYST. Each user has a main directory which has the same name as

his user name. Only SYST can create a user, but any user can create further directories. The system has a password facility so that only authorized people can access a particular directory, usually their own user directory. The procedure for connecting to the network is called 'logging on' and it is done by use of the command

```
*I AM nnn USER3 PASS
```

where nnn is the file server station selected, if other than the default station 254, USER3 is the name of the user, and PASS is the password for this user.

Unlike level 1, a user of the level 2 Econet system can use any of the stations on the network. It is not necessary to associate user names with the station identity as in level 1. The user name can consist of up to ten characters which must start with a letter. The password can be up to six characters long. The first time a user logs on to the system, the password will usually be blank.

If a user who is not known to the system tries to log on, the error message 'User not known' will be displayed. The system manager is responsible for creating new users, and for creating a user root directory for them. A user who does not have a directory can log on, but will not have any storage space allocated, and will only be able to read and use files on public access. At logon a user will normally have his own root directory as his selected directory. Extra security of passwords is provided in later versions of the Econet ROM in which the following sequence is permitted and will not reveal the password on the monitor screen

```
*I AM USER3 <RETURN>
PASS <RETURN>
```

In this case the second line will not be shown on the monitor screen. Use of the wrong password by a valid user will produce the error message 'Wrong password'. The user will not be logged on by the system and he must try again. At the end of a session at a station on the Econet the command

```
*BYE
```

logs off the user. The *BYE command is issued automatically during a logon so that only one user can be logged on at a station at one time. Until a user has logged into the network the only Econet file server command accepted is *I AM <user> <password>. The manager sets the number of stations that are to be served by a file server, and if an attempt is made to log on to a file server which has its full complement of users, the message 'Too many users' is sent.

The network stations on the Econet may include stations with DFS and disc drives. The range of standard DFS commands available on the Econet to non-DFS machines is nearly as extensive as those available on DFS machines. The commands on the DFS and Econet filing system (NFS) are as nearly compatible as possible. The DFS commands with no equivalent are *BACKUP, *BUILD, *COMPACT, *COPY, *DESTROY, *DRIVE,

*DUMP, *ENABLE, *LIST and *TYPE, but a system can have library utilities to provide some of these facilities, for example *BUILD, *DUMP and *TYPE. (See, for instance, *Networking with the BBC Microcomputer* by R G Napier, published by Prentice-Hall International.)

The commands that are different or modified are *ACCESS, *CAT, *DIR and *INFO. In addition to the DFS commands and those discussed for level 1, level 2 has the following additional commands available

```
*BYE  *CDIR  *DISCS  *EX  *LIB  *PASS  *SDISC
```

Table L.4 gives a summary of these commands and the modified DFS commands.

Table L.4 Level 2 Econet system commands.

Command	*Argument*	*Explanation*
*BYE		Logs the user off.
*CDIR	<dir name>	Creates a directory.
*DISCS		Displays the names of the discs currently in use.
*EX	<dir name>	Gives extensive information about files in directory <dir name>.
*I AM	nnn <name>	The logon command to station nnn by user <name>.
*INFO	<fsp>	Gives information about a file. It can not be used with wildcards as in the DFS.
*LIB	<dir name>	Selects <dir name> as the library directory.
*OPT	n,n	Chooses option 1,n selects file information display option. 4,n sets autostart option.
*PASS	"" <pass>	Sets user password to <pass>.
*PASS	<old> <new>	Changes user password from <old> to <new>.
*PS	<nnn>	Selects the printer server to be a station other than the default station 235. It is assumed the selected station is suitably equipped.
*SDISC	<disc name>	Changes the disc available to the user.

All the commands in Table L.1, except *DRIVE, are available to the level 2 user. The format for the commands *ACCESS, *INFO and *I AM are modified. The arguments of the file handling commands and *CAT are modified by the use of pathnames.

The majority of the modified commands that differ from their DFS counterparts can, in general be used as though there were no difference. The *ACCESS command is the most changed command due to the wider range of facilities offered by the network environment. The format for this command in the Econet situation is

```
*ACCESS <fsp> <owner access>/<public access> L
```

The presence of L indicates that the file is locked, to prevent its deletion. The owner access can be R for read and/or W for write, and the public access may be nil or read R. Thus

```
*ACCESS SAMPLE R/ L
```

sets the file in the currently selected directory called SAMPLE for read access by the owner, and at the same time prevents its deletion. Another format might be

```
*ACCESS MINEONLY RW/ L
```

or

```
*ACCESS EVERYONE RW/R L
```

The default setting of file access on creation of a file is RW/ . The access control is limited to either public access or no public access. There is no way of limiting access to a group of users.

The *PASS command is to enable a user to set a new value for a password. The format is

```
*PASS <oldpass> <newpass>
```

or, in the case of the initial setting

```
*PASS "" <pass>
```

The quotation marks are used to identify the empty string.

The *DISCS command enables a user to find out which discs the file server has loaded. A file server disc has a name to identify it. A user who needs a disc not currently loaded can then ask the system manager to load it, in place of one of those currently available. The command *SDISC allows a user to select the disc in the other drive for use, thus

```
*SDISC <name>
```

The user will need a directory, created by the system manager, on each disc on which he wishes to store programs or data.

The selection of a current library, by use of

```
*LIB <directory name>
```

is restricted to the disc selected by *SDISC, and so does not have the power of the similar DFS command. The library must be reselected after using *SDISC. The default library directory selected at logon and on change of disc by use of the command *SDISC is the directory 'Library'. 'Library' contains the system utilities given in Table L.2.

The *DIR command is used to select a directory on the currently selected disc. After a change of disc the user has his root directory as the currently selected directory. If any other is required then the command

```
*DIR NEWDIR
```

has to be used to make NEWDIR the currently selected directory. In order to select a directory such as 'Library' or that of another user, it is necessary to ascend to the root directory and then progress to the directory required by the appropriate route. The return to the user's own directory is made by

```
*DIR <RETURN>
```

without specifying the name or root. It will be found that on return to the user's directory by

```
*DIR $.<username>
```

that the user only has public access rights. To regain the normal rights of a user logged into their own directory it is necessary to use *DIR alone, logon again or to use the *SDISC command as though to change the selected disc.

A similar restriction and limitation to that for *LIB also applies to *CAT. Only directories on the selected disc can be catalogued and, unlike the DFS, a directory must be created before it is catalogued. A directory is created by

```
*CDIR <name>
```

This command can be issued by a user to create a sub-directory. Figure L.2 illustrates the hierarchical structure of the directories.

The format for the catalogue command is

```
*CAT <name>
```

If no name is given, the user's currently selected directory is catalogued. If the catalogue of another directory is required, then either that directory must be

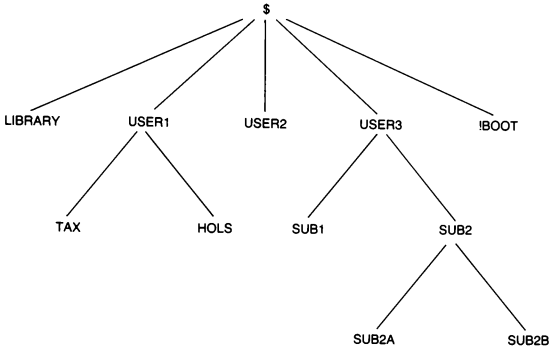


Figure L.2 The directory structure for level 2.

made the selected directory or the pathname of that directory must be given. For instance

```
*CAT $.Library
```

will catalogue the public access files in the directory 'Library' and

```
*CAT $.USER3.SUB2.SUB2A
```

will catalogue any files in USER3's sub-directory SUB2A which are on public access to any other user issuing the command.

The files may be referred to by pathnames in which the directory route to locate a file is given by a list of directory names, separated by the full stop. For example, the library file LSQ could be loaded by the command

```
LOAD "$.Library.LSQ"
```

without the need to change the currently selected directory (provided, of course, that the access control for public read is set). It is generally advisable to use the pathname, rather than to change directory to use a file in another directory, as it is then no longer necessary to use *DIR <RETURN> to regain owner rights in your own directory. The use of pathnames with files provides a convenient means to refer to files not in the currently selected directory. All the file handling commands can be used with pathnames as a

prefix provided that the access sought is allowed. (A useful tip if you are using a prefix frequently is to program it into a function key.)

The *INFO command is less flexible than that available with the DFS as it is no longer possible to use the wildcards * and # to give information about a group of files. It is limited to information about one file only, the first one found to fit the description. In the NFS environment, this command can also give further information about a directory.

The level 2 Econet command *EX compensates for the loss of the wildcard facility to some degree, as

```
*EX <directory name>
```

is equivalent to issuing a *INFO command for each of the files in the directory named.

The information displayed about files with *EX and *INFO is as follows for each file

```
NETFILE  FFFF1200  FFFF8023  000043
 WR/     14:4:85   000147
```

where

NETFILE	is the filename
FFFF1200	is the reload address
FFFF8023	is the execution address
WR/	is the access currently set
000043	is the file size
14:4:85	is the most recent date on which the file was written to
000147	is the System Internal Name (SIN)

The date is set each day by the system manager when the level 2 system is started.

The filenames on level 2 Econet can be up to ten characters long, and can contain any combination of letters and numbers. The following symbols may also be used

```
! % & = - ~ ^ | @ { [ £ _ + ; } ] < > ? /
```

but not the space character. No distinction is made between upper- and lower-case letters.

The *NOTIFY command can either be used with station identity or with user name. Thus both of the following messages, sent from station 121, are valid

```
*NOTIFY USER3 It is time for lunch <RETURN>
*NOTIFY 235 Please switch the printer on <RETURN>
```

The station at which USER3 is located will 'bleep' and display the message thus

```
--121: It is time for lunch
```

The message is not followed by a return so that the receiving user can delete it and continue with his own activity. The message at station 235 is

```
--121: Please switch the printer on
```

The number prefixing the message gives the station identity of the source of the message.

The range of network utilities available can be controlled by the network manager since he can change the access rights of the users to the files in the directory 'Library', which is the equivalent of the & directory in level 1. The utilities are programs that can be *RUN by the use of the abbreviation *<filename>. The user's own directory is scanned first for the file <filename>. If the file is not found in the owner's directory the 'library directory' is then scanned and if it is found then <filename> is *RUN. The library directory is either that last set by the *LIB command or if no change has been made it will be the directory 'Library' which is the default library directory set by the system during logon using the *I AM command. The system manager may choose to limit the range of utilities available to general users but this does not prevent their use by privileged users. This option has clear applications in class situations where a teacher may not want pupils to have the same access to the computers of other users that he can have.

The additional error messages of the level 2 Econet system are given in Table L.5.

Table L.5 Level 2 Econet error messages.

Message	*Explanation*
`Bad attribute`	*ACCESS has been used with an inappropriate access string.
`Bad password`	Whilst using *PASS, the new password is not acceptable.
`Bad string`	The file or pathname is too long or, the " symbol has been omitted.
`Broken dir`	The directory has been damaged. The system manager is responsible for keeping backup copies.
`Channel`	Corrupted memory (RAM) in user station. Log on again.
`Dir. full`	The directory on the selected disc is full.

Message	*Explanation*
`Dir. not empty`	An attempt has been made to delete a non-empty directory.
`Disc changed`	A change in file server disc has occurred.
`Entry locked`	It is necessary to use *ACCESS to unlock a file.
`EOF`	You have tried to read after the End Of File.
`Insufficient access`	You have tried to get access to a file for which you do not have sufficient access rights.
`Insufficient privilege`	There are certain commands only available to the system manager.
`Is a dir.`	An attempt to make incorrect use of a directory.
`Not found`	The file or directory specified is not where you expected.
`Not listening`	This can occur if you try to use the printer while it is busy.
`Not logged on`	The only command available to a user who is not logged on, is *I AM.
`Not open for update`	You are trying to write to a read-only file.
`Outside file`	Occurs with OPENIN in BASIC I. An attempt has been made to put the random access pointer past the file end.
`PW file not found`	This message may occur at logon if there is no password file on the file server.
`Rename across discs`	The *RENAME command does not allow a file to be moved between different directories on different discs.
`Too many open files`	In general data file access, you have used all the channels available.
`Too many users`	The system manager sets the maximum number of users for the file server.
`Types don't match`	You have tried to save a file to a directory, or similar.
`User not known`	The system manager creates users.

Message	*Explanation*
`Who are you?`	Reminder that *I AM is the only command for non logged on users.
`Wrong password`	An attempt to log on with an incorrect password.

This table does not include errors common to level 1 and the DFS.

Appendix M

USING THE DISCZAP PROGRAM

Before discussing the DISCZAP option in the ZAP program, a word of warning should be given. When writing sectors directly to a disc, it is very easy to corrupt an entire disc irretrievably. Before working on a disc, therefore, *it is essential that you make a backup copy of the disc.*

When the DISCZAP option is chosen from program ZAP (Section 10.5), the command P(age) is replaced by three other commands: Rt,s – read track t, sector s; Wt,s – write track t, sector s; and Vn – select drive n.

The read and write commands must be issued in the form

```
R15,9 and W25,0
```

with track and sector numbers in *decimal* and separated by a comma because the numbers are collected by a two-variable INPUT command.

A sector is split into two 128-byte display pages, which can be examined using the cursor keys, with or without SHIFT. The program always begins by reading track 0, sector 0 – the first of the two catalogue sectors. Adjacent sectors can be read simply by scrolling the cursor off the top or bottom of the two pages of display. Larger jumps can be made with the R command. A sector can only be written back to disc with the W command, and the program checks first that you really do intend to write a sector.

The current track and sector is displayed at the top of the display, but you can write a sector back to a different place on the disc if you have a reason to do so. Initially the current drive remains in force, and only when you issue a command such as V1 is the drive changed and the value displayed at the top of the screen.

The remaining commands are the same as for MEMZAP.

HOW THE DISCZAP PROGRAM WORKS

Reading a sector from disc, and writing it back, is carried out by OSWORD calls &7F at lines 6490 and 6720. These involve calls to address &FFF1 with

&7F in the accumulator (via resident integer variable A%). X% and Y% contain low and high bytes of the address of a parameter block in memory – in this case the section of zero page starting at &70.

The parameter block is set up partly on line 160 and partly on lines 6450 or 6680. The parameters are, in order: the drive number; the address in memory for the source or destination of the sector (4 bytes, low byte first); 3; &53 for read or &4B for write; track; sector; &21; error number on return from the call (0 for a successful call).

The program uses page &7B of memory (immediately below the Mode 7 screen display area) for storing the sector and sets HIMEM to address &7B00, below this buffer area.

The R and W commands set the seventh byte of the parameter block to &53 or &4B respectively, track and sector are set via the variables TRACK and SECTOR, and then the call to &FFF1 is made, checking afterwards that there is no error from the call. If an error is detected the procedures try up to ten times before giving up.

The V command to change drive simply alters variable DRIVE which is used in the parameter list. Remaining commands work as they do for MEMZAP, moving the cursor or altering the specific page of memory being used as the disc buffer. Obviously the Page command is inappropriate for DISCZAP. When the cursor runs off either end of the page of memory representing the sector, this is trapped and the preceding or following sector is read from disc by a call to the 'read sector' subroutine.

THE STRUCTURE OF THE DFS CATALOGUE SECTORS

In order to make use of the DISCZAP utility, it is necessary to be able to find your way around the sectors of a disc, and the key to this is the two sectors containing the catalogue information. These are sectors 0 and 1 on track 0. Sector 0 is split up into 8 byte segments, and each segment except the first may contain the name of a file on the disc (hence the 31 file limit for the DFS). In corresponding positions in sector 1 is the catalogue information for each file. This is broadly similar to the information given by the *INFO command.

The precise use of the two catalogue sectors is as follows

Sector 0

bytes 0 to 7	First 8 characters of the disc title
bytes 8 to 14	Name of last file on disc (maximum 7 characters)
byte 15	File directory letter and lock status
bytes 16 to 22	Name of penultimate file
...	
...	
...	

Sector 1

bytes 0 to 3	Last 4 characters of disc title
byte 4	Number of times the disc has been accessed for writing
byte 5	8×number of entries in catalogue
byte 6 (high nibble)	Startup option set by *OPT 4,n
byte 6 (low nibble) and byte 7	Number of sectors on disc (&190 for 40 track discs, &320 for 80 track)
bytes 8 to 15	Information on the last file on the disc, as follows
bytes 8 and 9	Load address of file
bytes 10 and 11	Execution address of file (&801F or &8023 for BASIC programs)
bytes 12 and 13	File length in bytes
byte 14	High bits of each number
byte 15	Sector address of start of file (in hex)
bytes 16 to 23	Information on the penultimate file
...	
...	
...	

Three of the items require further explanation. The eighth byte of each file name contains the directory letter for the file in the first seven bits, while the high bit is set to indicate a file that is locked or zero if the file is not locked.

In sector 1 the eighth byte for each file contains the sector address of the file, that is the number of sectors from the start of the disc. To make use of this figure, it will be necessary to convert it into decimal and hence calculate the track and sector values before using DISCZAP.

The seventh byte contains spillover bits from the other four items. This is because the addresses and file length are stored as 18-bit numbers, and the sector address as a 10-bit number. The two high bits of each number are stored in byte seven as follows: bits 0 and 1 – sector address; bits 2 and 3 – load address; bits 4 and 5 – file length; bits 6 and 7 – execution address. These bits will usually be zero except for the sector address, so the byte will effectively act as the high byte of the sector address.

Figure M.1, taken from DISCZAP, shows sectors 0 and 1 of the BBC utility disc, and many of the features mentioned above can be seen.

With an understanding of the way in which catalogue information is stored, you should be able to find your way around the sectors on a disc, and read or modify files direct on disc. Perhaps most valuable of all, used with care, is the editing of the catalogue sectors themselves. To give just one example, it is easy with the first version of *Wordwise* to wipe out a file by 'saving' it with nothing in memory when you intended to load it. All that will actually have happened on the disc is that the length will be marked as zero in

```
Sector 0

00 55 74 69 6C 69 74 79 00 Utility.
08 4D 45 4D 4F 44 20 20 A4 MEMOD  $
10 44 43 4F 4E 56 20 20 A4 DCONV  $
18 56 45 52 49 46 59 20 A4 VERIFY $
20 46 4F 52 4D 38 30 20 A4 FORM80 $
28 46 4F 52 4D 34 30 20 A4 FORM40 $
30 21 42 20 20 20 20 20 24 !B     $
38 42 49 4F 52 54 48 4D 57 BIORTHMW
40 50 48 4F 54 4F 20 20 57 PHOTO  W
48 50 4F 45 4D 20 20 20 57 POEM   W
50 21 42 4F 4F 54 20 20 24 !BOOT  $

Sector 1

00 00 00 00 00 06 D8 31 90 .....X1.
08 00 00 00 00 F1 20 01 4C ....q .L
10 00 19 1F 80 C9 01 01 4A ....I..J
18 00 28 00 28 00 02 01 48 .(.(...H
20 00 28 04 28 00 03 01 45 .(.(...E
28 00 28 00 28 00 03 01 42 .(.(...B
30 00 19 1F 80 BA 02 01 3F ....:..?
38 00 19 1F 80 1D 19 01 25 .......%
40 00 19 00 19 00 10 01 15 ........
48 00 19 1F 80 B1 25 00 EF ....1%.o
50 00 30 00 30 0B 00 00 BB .0.0...;
```

Figure M.1 DISCZAP display for the BBC utility disc.

the catalogue sectors, and the first sector may have been overwritten. The rest of the old file can be tracked down on the disc, and recovered by altering the length in the catalogue back to its full extent.

(The alternative disaster under *Wordwise*, obliterating the text in memory by 'loading' a nonexistent file when you meant to save it, can be recovered with MEMZAP, by tracking down the text which will still exist somewhere in memory, and saving it with *SAVE.)

Appendix N

ANSWERS TO EXERCISES

Exercise 2.1

```
10 INPUT "WHAT IS THE CAPITAL OF FRANCE",A$
15 M=0
20 IF A$="PARIS" THEN GOTO 60
25 M=M+1
27 IF M=2 THEN 80
30 PRINT "WRONG, TRY AGAIN"
40 INPUT A$
50 GOTO 20
60 PRINT "CORRECT"
70 END
80 PRINT "NO, THE ANSWER IS PARIS"
90 END
```

Exercise 2.2

```
10 REM SUM OF THE FIRST N WHOLE NUMBERS
20 INPUT "HOW MANY NUMBERS",N
30 SUM=0
40 FOR J%=1 TO N
50 SUM=SUM+J%
60 NEXT J%
70 PRINT "SUM OF THE FIRST ";N;" WHOLE NUMBERS =";SUM
80 END
```

Exercise 2.3

```
10 REM MASTERMIND
20 X=RND(10)
30 PRINT "WHAT NUMBER AM I THINKING OF?"' "(BETWEEN 1 AND 10)"
40 REPEAT
```

```
50 INPUT GUESS
60 UNTIL GUESS=X
70 PRINT "CORRECT"
```

Exercise 2.4

```
 10 REM GUESS THE NUMBER
 20 X=RND(1000)
 30 PRINT "WHAT NUMBER AM I THINKING OF?"' "(BETWEEN 1 AND
    1000)"
 35 I=0
 40 REPEAT
 42 IF I=40 THEN PRINT "GIVE UP! THE ANSWER IS ";X: GOTO 80
 45 I=I+1
 50 INPUT GUESS
 52 IF GUESS>X THEN PRINT "TOO HIGH"
 54 IF GUESS<X THEN PRINT "TOO LOW"
 60 UNTIL GUESS=X
 70 PRINT "CORRECT"
 80 PRINT "YOU TOOK ";I;" GUESSES"
 90 IF I>10 THEN PRINT "YOU HAVEN'T DISCOVERED THE SYSTEM YET"
100 IF I<=10 THEN PRINT "CONGRATULATIONS"
110 END
```

Exercise 2.5

```
*KEY0 LISTO 0 |M |O |M
*KEY1 LISTO 7 |M |N LIST |M
```

Exercise 2.6

```
 10 REM PRODUCT OF 10 NUMBERS
 20 DIM NUM(10)
 30 FOR J=1 TO 10
 40 READ NUM(J)
 50 NEXT J
 60 PRODUCT=1
 70 FOR J=1 TO 10
 80 PRODUCT=PRODUCT*NUM(J)
 90 NEXT J
100 PRINT "PRODUCT = ";PRODUCT
110 END
120 DATA 2,5,3,7,1,4,6,9,10,8
```

Exercise 3.1

```
 10 REM HYPERBOLIC SIN AND COS
 20 REPEAT
 30 INPUT "VALUE OF X (USE 0 TO FINISH)",X
```

```
 40 C=FN_hcs(X)
 50 S=FN_hsn(X)
 60 Y=C*C-S*S: REM SHOULD EQUAL ONE
 70 PRINT "COSH(X)="C
 80 PRINT "SINH(X)="S
 90 PRINT "COSH SQUARED - SINH SQUARED ="Y
 95 PRINT
100 UNTIL X=0
105 END
110 DEF FN_hcs(X)=(EXP(X)+EXP(-X))/2
120 DEF FN_hsn(X)=(EXP(X)-EXP(-X))/2
```

Alternatively, we can print the results in columns

```
 10 REM HYPERBOLIC SIN AND COS
 15 @%=&20509: REM @% IS EXPLAINED IN FULL IN CHAPTER 11. THIS
    IS SETTING THE FORMAT TO 5 DECIMAL PLACES, SO THAT THE
    OUTPUT IS IN NEAT COLUMNS
 18 PRINT: PRINT " X COSH SINH TEST=1?"
 19 PRINT
 20 FOR X=0 TO 2.01 STEP 0.1
 30 C=FN_hcs(X)
 40 S=FN_hsn(X)
 50 Y=C*C-S*S: REM SHOULD EQUAL ONE
 60 PRINT ;X,C,S,Y
 70 NEXT X
 80 END
 90 DEF FN_hcs(X)=(EXP(X)+EXP(-X))/2
100 DEF FN_hsn(X)=(EXP(X)-EXP(-X))/2
```

Exercise 3.2

```
 10 REM FAHRENHEIT/CENTIGRADE CONVERSION
 20 INPUT "GIVE A TEMPERATURE IN FAHRENHEIT",TF
 30 TC=FN_CENT(TF)
 40 PRINT "TEMPERATURE IN CENTIGRADE ="TC
 50 INPUT "GIVE A TEMPERATURE IN CENTIGRADE",TC
 60 TF=FN_FAHR(TC)
 70 PRINT "TEMPERATURE IN FAHRENHEIT="TF
 80 END
 90 DEF FN_CENT(T)
100 =(T-32)/1.8
110 DEF FN_FAHR(T)
120 =T*1.8+32
```

Exercise 3.3

Modify Example 3.3 as follows

```
110 DEF PROC_delay(CENTISEC)
160 ENDPROC
190 PROC_delay(RND(150))
```

Further modifications to eliminate the formal parameter are

```
110 DEF PROC_delay
150 UNTIL (100+START)<TIME
160 ENDPROC
190 PROC_delay
```

Exercise 3.4

A suitable procedure is

```
100 DEF PROC_LOWHIGH
110 REM finds the largest and smallest numbers in array NOS of
    10 numbers
120 LOW=NOS(1): HIGH=NOS(1)
130 FOR J=2 TO 10
140 IF NOS(J)>HIGH THEN HIGH=NOS(J)
150 IF NOS(J)<LOW THEN LOW =NOS(J)
160 NEXT J
170 ENDPROC
```

which can be tested with the following program

```
10 REM PROGRAM TO TEST PROC_LOWHIGH
20 DIM NOS(10)
25 PRINT "TYPE IN 10 NUMBERS, ONE PER LINE"
30 FOR J=1 TO 10
40 INPUT NOS(J)
50 NEXT J
60 PROC_LOWHIGH
70 PRINT
80 PRINT "LOWEST="LOW
90 PRINT "HIGHEST="HIGH
99 END
```

Exercise 3.5A

A suitable procedure is

```
100 DEF PROC_PRIME(RANGE)
110 PRIMES(1)=1
120 LOCAL N,X,Y,Z
130 N=1
140 Y=2
150 REPEAT
160 N=N+1
170 PRIMES(N)=Y
```

```
180 REPEAT
190 Y=Y+1
200 X=1
210 REPEAT
220 X=X+1
230 Z=INT(Y/X)
240 UNTIL (Z*X=Y OR X*X>Y)
250 UNTIL X*X>Y
260 UNTIL N=RANGE
270 ENDPROC
```

Exercise 3.5B

A suitable program to test the procedure is

```
10 INPUT "No. of prime numbers ",N
20 DIM PRIMES(N)
30 PROC_PRIME(N)
40 SUM=0
50 FOR J=1 TO N
60 SUM=SUM+PRIMES(J)
70 NEXT J
80 PRINT "Sum of first ";N;" primes = ";SUM
90 END
```

Exercise 3.6

```
 10 REM ON...GOTO DEMONSTRATION
 20 REPEAT
 30 INPUT "Give the time on a 24 hour clock",ZTIME
 40 ON ZTIME GOTO50,50,50,50,50,50,50,70,70,70,70,90,90,
    90,90,90,110,110,110,110,130,130,130 ELSE 150
 50 GS$="MORNING"
 60 GOTO 140
 70 GS$="DAY"
 80 GOTO 140
 90 GS$="AFTERNOON"
100 GOTO 140
110 GS$="EVENING"
120 GOTO 140
130 GS$="NIGHT"
140 PRINT "GOOD ";GS$
150 UNTIL ZTIME>24
160 PRINT "GOODBYE!"
```

Exercise 3.7

```
10 X=TIME
20 T$=INT(X/100)
```

```
30 TM=INT(TS/60)
40 TH=INT(TM/60)
50 TD=INT(TH/24)
60 PRINT "Time ";TD;" days ";TH;" hours ";TM MOD 60; " minutes
   ";TS MOD 60;" second"
```

Exercise 3.8A

```
 10 REM PERPETUAL CALENDAR
 20 DIM MON$(12),DAY(12)
 30 MODE 0
 40 INPUT "For which year do you want a calendar ",YEAR
 50 IF YEAR<1800 THEN CLS: PRINT "Remember the year must be
    post 1800": GOTO 40
 60 FOR I=1 TO 12: READ MON$(I),DAY(I): NEXT
 70 DAY1=(((YEAR-1800)*365+(YEAR-1800) DIV 4)-(YEAR DIV 100 -
    YEAR DIV 400 -14)+3) MOD 7
 80 IF ((YEAR DIV 4)*4=YEAR) AND ((YEAR DIV 400)*400 <>YEAR)
    THEN DAY(2)=29: DAY1=(DAY1+6) MOD 7 ELSE DAY(2)=28
100 CLS
110 PRINT TAB(38,0);YEAR
120 FOR J=0 TO 3
125 I=J*3+1: K=J*8+1: L=I
130 PRINT ;TAB(11,K);MON$(I);TAB(36,K);MON$(I+1);TAB(61,K);
    MON$(I+2)
140 PRINT STRING$(3,"     Su Mo Tu We Th Fr Sa")
150 REPEAT
160 PROC_DATES(L,J,DAY1)
170 DAY1=(DAY1+DAY(L)) MOD 7
180 L=L+1
190 UNTIL L=I+3
200 NEXT
210 PRINT TAB(0,31);
400 END
500 DEF PROC_DATES(LD,JD,SDAY)
520 DATE=1
530 CDAY=SDAY
540 REPEAT
550 PRINT ;TAB(((LD-1) MOD 3)*25+(CDAY MOD 7)*3+5,JD*8+ (CDAY
    DIV 7) MOD 5 +3);DATE;
560 DATE=DATE+1
570 CDAY=CDAY+1
580 UNTIL DATE-1=DAY(LD)
600 ENDPROC
900 DATA January,31,February,28,March,31,April,30,May,31,
    June,30,July,31,August,31,September,30,October,31,
    November,30,December,31
```

Exercise 3.8B

This modified solution uses an array to produce the screen display. This modification allows the calendar to be sent to a printer, if after use of the appropriate printer calls the section of program at the end is implemented by the use of GOTO 2000 in immediate mode.

```
 10 REM PERPETUAL CALENDAR
 20 DIM MON$(12),DAY(12),SCRPO$(20,19)
 25 FOR I=0 TO 19: FOR J=0 TO 20: SCRPO$(J,I)=" ": NEXT: NEXT
 30 MODE 0
 40 INPUT "For which year do you want a calendar ",YEAR
 50 IF YEAR<1800 THEN CLS: PRINT "Remember the year must be
    post 1800": GOTO 40
 60 FOR I=1 TO 12: READ MON$(I),DAY(I): NEXT
 70 DAY1=(((YEAR-1800)*365+(YEAR-1800) DIV 4)-(YEAR DIV 100 -
    YEAR DIV 400 -14)+3) MOD 7
 80 IF ((YEAR DIV 4)*4=YEAR) AND ((YEAR DIV 400)*400 <>YEAR)
    THEN DAY(2)=29: DAY1=(DAY1+6) MOD 7 ELSE DAY(2)=28
100 CLS
110 PRINT TAB(38,0);YEAR
120 FOR J=0 TO 3
125 I=J*3+1: K=J*8+1: L=I
130 PRINT ;TAB(11,K);MON$(I);TAB(36,K);MON$(I+1);TAB(61,K);
    MON$(I+2)
140 PRINT STRING$(3,"     Su Mo Tu We Th Fr Sa")
150 REPEAT
160 PROC_DATES(L,J,DAY1)
170 DAY1=(DAY1+DAY(L)) MOD 7
180 L=L+1
190 UNTIL L=I+3
200 NEXT
400 END
500 DEF PROC_DATES(LD,JD,SDAY)
520 DATE=1
530 CDAY=SDAY
540 REPEAT
545 X=((LD-1) MOD 3)*7+(CDAY MOD 7): Y=JD*5+(CDAY DIV 7)
    MOD 5
547 IF DATE<10 THEN SCRPO$(X,Y)=" "+STR$(DATE) ELSE
    SCRPO$(X,Y)=STR$(DATE)
550 PRINT ;TAB(((LD-1) MOD 3)*25+(CDAY MOD 7)*3+5,JD*8+ (CDAY
    DIV 7) MOD 5 +3);DATE;
560 DATE=DATE+1
570 CDAY=CDAY+1
580 UNTIL DATE-1=DAY(LD)
600 ENDPROC
900 DATA January,31,February,28,March,31,April,30,May,31,
    June,30,July,31,August,31,September,30,October,31,
```

```
     November,30,December,31
2000 PRINT ;TAB(38);YEAR: PRINT
2010 FOR K=0 TO 3: M=K*3+1
2020 PRINT;TAB(11);MON$(M);TAB(36);MON$(M+1);TAB(61);
     MON$(M+2)
2030 PRINT STRING$(3,"     Su Mo Tu We Th Fr Sa")
2040 FOR L=0 TO 4: I=K*5+L
2050 FOR J=0 TO 20
2060 IF (J MOD 7)=0 THEN PRINT ;"    ";
2070 PRINT ;" ";SCRPO$(J,I);
2080 NEXT
2090 PRINT
2100 NEXT
2110 PRINT
2120 NEXT
```

Exercise 3.9A

The simple solution is to modify Example 3.11 as shown below

```
 10 DIM N(6),M(3)
 20 FOR L = 1 TO 20
 30 FOR I = 1 TO 6: N(I)=I: NEXT
 40 PROC_perm(3)
 50 PRINT ;L;"   ";M(1);"    ";M(2);"    ";M(3)
 60 NEXT L
 70 END
 80 DEF PROC_perm(I)
 90 IF I<>1 THEN J=RND(I+3): M(I)=N(J): N(J)=N(I+3):
    PROC_perm(I-1) ELSE M(I)=N(RND(I+3))
100 ENDPROC
```

Exercise 3.9B

A more widely applicable solution is the following program, that can be used to select a random set of N numbers from a given range R.

```
  5 INPUT "Combination of N from R give N and R ",N,R
 10 DIM N(R),M(N)
 20 FOR L = 1 TO 20
 30 FOR I = 1 TO R: N(I)=I: NEXT
 40 PROC_perm(N)
 50 PRINT ;L;
 55 FOR J=1 TO N: PRINT"   ";M(J);: NEXT
 57 PRINT
 60 NEXT L
 70 END
 80 DEF PROC_perm(I)
```

```
 90 IF I<>1 THEN J=RND(I+R-N): M(I)=N(J): N(J)=N(I+R-N):
    PROC_perm(I-1) ELSE M(I)=N(RND(I+R-N))
100 ENDPROC
```

Exercise 4.6

```
10 PRINT "INPUT ANY STRING"
20 INPUT A$
30 L=LEN(A$)
40 B$=LEFT$(A$,INT(L/2))
50 C$=RIGHT$(A$,L-INT(L/2))
60 PRINT B$+" EXTRA "+C$
70 REM OR PRINT B$;" EXTRA ";C$ WOULD DO
```

Exercise 4.7

```
10 PRINT "INPUT ANY STRING"
20 INPUT A$
30 FOR J=LEN(A$) TO 1 STEP -1
40 PRINT MID$(A$,J,1);
50 NEXT J
60 PRINT
```

Exercise 4.9

Change line 130 of Exercise 4.8 to

```
130 UNTIL J=0 OR (LEN(A$)-LEN(B$))<(L-1)
```

Exercise 4.11

```
 10 PRINT "INPUT A NUMBER IN FIXED POINT OR EXPONENT FORM"
 20 INPUT A$
 30 L=LEN(A$): D=0
 40 FOR J=1 TO L
 50 IF MID$(A$,J,1)="E" THEN GOTO 110
 60 IF MID$(A$,J,1)="." THEN D=J
 70 NEXT J
 80 B$=RIGHT$(A$,L+1-D)
 90 B=VAL(B$)
100 IF D>0 THEN PRINT 'B
110 PRINT '"END"
```

Exercise 4.13

CHR$(7) produces a bleep since this is the BELL character
CHR$(8) moves the cursor to the left one space
CHR$(10) moves the cursor down one line
CHR$(12) clears the screen and homes the cursor to top left

Exercise 4.14

The following program carries out all the tasks set in Exercise 4.14

```
 10 PRINT "INPUT ANY CHARACTER"
 20 INPUT A$
 30 PRINT "ASCII CODE OF ";A$;" IS ";ASC(A$)
 40 PRINT
 50 PRINT "INPUT ANY ASCII CODE (33 TO 255)"
 60 INPUT A
 70 PRINT "THE CHARACTER ";CHR$(A);" HAS ASCII CODE ";A
 80 PRINT
 90 TIME=0: REPEAT: UNTIL TIME>=300
100 FOR J=33 TO 255
110 PRINT J;"  ";CHR$(J)
120 TIME=0: REPEAT: UNTIL TIME>=20
130 NEXT J
```

Exercise 4.15

Modify Exercise 4.7 by

```
25 DIM A$(LEN(A$))
30 FOR J=1 TO LEN(A$)
40 A$(J)=MID$(A$,J,1)
60 FOR J=LEN(A$) TO 1 STEP -1: PRINT A$(J);: NEXT J
70 PRINT
```

Exercise 5.2

```
10 PRINT "TYPE YOUR EXPRESSION AS A FUNCTION"
20 INPUT "OF X: ",EX$
30 PRINT "TYPE UPPER AND LOWER LIMITS"
40 INPUT "OF X: ",A,B
50 FOR X=A TO B STEP (B-A)/10
60 PRINT X;"  ";EVAL(EX$)
70 NEXT X
```

Exercise 5.3

```
10 INPUT "TYPE THE AMOUNT, IN PENCE: "AMT%
20 PDS%=AMT% DIV 100
30 PENCE%=AMT% MOD 100
40 PRINT "AMOUNT IS £";PDS%;" ";PENCE%;"p"
```

Exercise 5.5

Line 170 can be rewritten as

```
170 SCORE(PLAYER)=SCORE(PLAYER)-10*(ACE(PLAYER)>0 AND
    SCORE(PLAYER)<=11)
```

with a similar line for 260.

Exercise 6.5

In Example 6.4, change line 70 to

```
70 VDU 29,640;608;
```

and delete (or replace by zero) X0 and Y0 from lines 80, 90, 130 and 170. Proceed similarly for the other programs.

Exercise 6.6

Change line 30 to, say

```
30 VDU 28,3,9,16,2
```

Exercise 6.8

A greek phi can be created by

```
VDU 23,225,8,8,28,42,28,8,8,0
```

Exercise 6.9

The following VDU statements define, in order, an open and closed square, an open and close triangle and an open and closed circle .

```
VDU 23,230,248,136,136,136,248,0,0,0
VDU 23,231,248,248,248,248,248,0,0,0
VDU 23,232,32,80,80,136,248,0,0,0
VDU 23,233,32,112,112,248,248,0,0,0
VDU 23,234,112,136,136,136,112,0,0,0
VDU 23,235,112,248,248,248,112,0,0,0
```

Exercise 7.1

The program contains the following errors

line 20	should be DIM(3)
line 40	should not be a REM
line 200	should not contain a minus sign
line 220	closing bracket omitted
line 230	should be T, not T$
line 230	missing colon
line 390	must return to line 30, not 20

Exercise 9.3

```
10 INPUT "TYPE THE NUMBER OF CARS: ",N
20 CH%=OPENOUT("CARFILE")
30 FOR J=1 TO N
```

```
40 PRINT "TYPE THE NAME OF THE CAR"
50 INPUT CAR$
60 PRINT#CH%,CAR$
70 NEXT J
80 CLOSE#CH%
```

Exercise 9.4

```
10 PRINT "LIST OF CARS": PRINT
20 CH%=OPENIN("CARFILE")
30 REPEAT
40 INPUT#CH%,CAR$
50 PRINT CAR$
60 UNTIL EOF#CH%
70 CLOSE#CH%
```

Exercise 9.5

Modify Example 9.5 by the following addition

```
65 IF LEN(ITNM$)+LEN(TPE$)+9>32 THEN PRINT "YOUR RECORD IS TOO
   LONG: RE-INPUT": GOTO 40
```

Exercise 9.6

Example program 9.7 requires the following addition

```
1015 IF RECNO*32>EXT#CH% THEN PRINT "THAT RECORD NUMBER DOES
     NOT EXIST": GOTO 1080
```

Exercise 9.7

Modify Example 9.7 further, as follows

```
1075 PROC_modify(RECNO)
3000 DEF PROC_modify(RECNO)
3010 INPUT "DO YOU WANT TO CHANGE THE QUANTITY (Y/N)",YN$
3020 IF LEFT$(YN$,1)="N" THEN ENDPROC
3030 INPUT "TYPE THE NEW QUANTITY: ",QU%
3040 PTR#CH%=RECNO*32
3050 PRINT#CH%,ITNM$,TPE$,QU%
3060 ENDPROC
```

Exercise 9.8

```
10 CH%=OPENIN("INVENT")
20 C=0
30 REPEAT
40 C=C+1
```

```
50 PRINT ;~BGET#CH%;" ";
60 IF C=8*INT(C/8) THEN PRINT
70 UNTIL EOF#CH%
80 CLOSE#CH%
```

Exercise 9.9

Modify the program from Exercise 9.3 as follows

```
52 L%=LEN(CAR$)
54 IF L%>10 THEN PRINT "NOT MORE THAN 10 CHARACTERS":GOTO 50
56 CAR$=CAR$+STRING$(10-L%," ")
```

Exercise 9.10

```
10 CH%=OPENIN("CARFILE")
20 PRINT "WHICH CAR DO YOU REQUIRE (1,2,3,...)"
30 INPUT N
40 PTR#CH%=(N-1)*12
50 INPUT#CH%,CAR$
60 PRINT "CAR ";N;" IS ";CAR$
70 CLOSE#CH%
```

Exercise 9.11

The ASCII file should have the form

```
*KEY 0 RUN |M
*KEY 1 LIST |M
*KEY 2 AUTO |M
*KEY 3 RENUMBER |M
*KEY 4 *RENAME
*KEY 5 *DELETE
```

Exercise 9.13

Delete lines 10 to 310 of Example 4.10, and add or modify the remaining lines as follows

```
300 DEF PROC_textprint(L)
310 LOCAL J,K,M
385 M=0
390 REPEAT
395 M=M+1
405 IF T$(M+K)=CHR$(13) THEN J=M
410 UNTIL M>=J-1
460 ENDPROC
```

Then renumber the resulting procedure to start at 2000 instead of 300. Append the procedure to Example 9.10, and make further changes to the program, as follows

```
525 DIM T$(3500)
700
760 L=L+1
770 T$(L)=CH$
780 UNTIL EOF#CH% OR L>=3500
784 IF (OP% AND 4)=4 THEN VDU 2
786 PROC_textprint(L)
```

A minor problem with this program is that, because it now stores the text as single characters in the string array T$, it is very wasteful of memory. Much larger files could be read if the data was poked directly into memory with the indirection operator instead of into an array. This can be achieved by the following further changes to the program above. As a further refinement, a choice of page widths is offered, to accommodate output to a printer, or use in Mode 0 or Mode 3.

```
525 BA=&2500
682 PRINT "What page width do you require?"
684 INPUT PGW
770 ?(BA+L)=ASC(CH$)
780 UNTIL EOF#CH% OR L>&5000: REM REDUCE VALUE OF &5000 IF
    NOT IN MODE 7
782 ?(BA+L+1)=0
784 IF (OP% AND 4)=4 THEN VDU 2
786 PROC_textprint(L,BA)
2000 DEF PROC_textprint(L,BA)
2010 LOCAL J,K,M,CH$
2040 J = PGW+2
2060 J = J-1: CH$=CHR$(?(BA+J+K))
2070 UNTIL CH$ = " " OR CH$ = CHR$(0) OR (CH$ = "-" AND
   J<PGW+1) OR J=0
2080 IF J = 0 THEN J = PGW: REM WORD TOO LONG TO FIT
2115 CH$=CHR$(?(BA+M+K))
2120 PRINT CH$;
2130 IF CH$=CHR$(13) THEN J=M
2145 CH$=CHR$(?(BA+J+K))
2150 IF CH$ <> " " THEN PRINT CH$;
2160 IF J<PGW OR CHR$(?(BA+PGW+K)) = " " OR CHR$(?(BA+PGW+K))
    = "" THEN PRINT
```

Exercise 10.1

```
10 FOR J=7 TO 0 STEP -1
20 MODE J
30 PRINT "PAGE = ";PAGE
40 PRINT "TOP = ";TOP
50 PRINT "HIMEM = ";HIMEM
60 PRINT "LOMEM = ";LOMEM
```

```
 70 PRINT "PROGRAM SIZE = ";(TOP-PAGE)
 80 PRINT "FREE SPACE = ";(HIMEM-TOP)
 90 PRINT "PRESS ANY KEY TO CONTINUE"
100 A$=GET$
110 NEXT J
```

Exercise 10.2

PAGE should have the value &2300.

Exercise 10.6

$M=A$ means 'load the string stored in variable A$ into memory starting from address M'.
M$=$A means 'set variable M$ equal to the string (hopefully) stored in memory starting from address A'.

Exercise 10.8

```
10 PRINT "TYPE THE STARTING ADDRESS FOR STRING"
20 INPUT ADDR$
25 ADDR=EVAL(ADDR$)
30 PRINT "TYPE THE STRING"
40 INPUT A$
50 FOR J=0 TO LEN(A$)-1
60 ADDR?J=ASC(MID$(A$,J+1,1))
70 NEXT J
75 J=LEN(A$)
80 ADDR?J=13
90 PRINT "STRING WAS: ";$ADDR
```

Exercise 10.17

If the line P%=start% is not inside the FOR...NEXT loop, the second assembly will start at the end of the first assembly. Note the different assembly addresses for the two passes.

Exercise 11.1

The program from Exercise 4.14 can be modified as follows

```
20 A$=GET$
```

Exercise 11.2

```
10 MODE 6
20 VDU 23,1,0;0;0;0;
30 *FX 4,1
40 ST$="*"
50 CH=20: CV=12
```

```
 60 PRINT TAB(CH,CV);ST$;
 70 REPEAT
 80 A=ASC(GET$)-135
 90 PRINT TAB(CH,CV);" ";
100 IF A>0 AND A<3 THEN CH=CH+2*A-3
110 IF A>2 AND A<5 THEN CV=CV-2*A+7
120 IF CV>24 THEN CV=24
130 IF CV<0 THEN CV=0
140 IF CH>39 THEN CH=39
150 IF CH<0 THEN CH=0
160 PRINT TAB(CH,CV);ST$;
170 UNTIL A=0
```

Exercise 11.3

Modify the program from Example 11.1 as follows

```
 30 PRINT "PRESS THE KEY SPECIFIED"
105 K=64+RND(26)
120 PRINT TAB(15,12);"PRESS THE KEY: ";CHR$(K)
160 UNTIL INKEY(0)=K
```

Exercise 11.4

Modify the program from Example 11.2 as follows

```
 70 IF INKEY(-99) THEN PROC_wait
1030 UNTIL NOT INKEY(-99)
```

Exercise 11.8

A suitable procedure is

```
1010 DEF PROC_table
1020 LOCAL J,K
1030 FOR J=1 TO 5
1040 PRINT TAB(7*J-7);HEAD$(J);
1050 NEXT J
1060 PRINT ''
1070 FOR K=1 TO 10
1080 FOR J=1 TO 5
1090 PRINT TAB(7*J-7);A$(K,J);
1100 NEXT J
1110 PRINT
1120 NEXT K
1130 ENDPROC
```

Exercise 11.9

The following procedure will provide the required display

```
1000 DEF PROC_clock(HOUR1,MIN1,SEC1,HOUR2,MIN2,SEC2)
1010 PRINT TAB(0,7);SPC(16);
1020 PROC_display(HOUR1,MIN1,SEC1)
1030 PRINT SPC(16)
1040 PRINT TAB(0,15);SPC(16);
1050 PROC_display(HOUR2,MIN2,SEC2)
1060 PRINT SPC(16)
1070 ENDPROC
2000 DEF PROC_display(HOUR,MIN,SEC)
2010 LOCAL HOUR$,MIN$,SEC$
2020 HOUR$=STR$(HOUR): IF HOUR<10 THEN HOUR$="0"+HOUR$
2030 MIN$=STR$(MIN): IF MIN<10 THEN MIN$="0"+MIN$
2040 SEC$=STR$(SEC): IF SEC<10 THEN SEC$="0"+SEC$
2050 PRINT HOUR$;":";MIN$;":";SEC$;
2060 ENDPROC
```

Exercise 11.10

The following program should be used in conjunction with PROC_formprint from Example 11.8.

```
  5 MODE 0
 10 FOR J=1 TO 20
 20 I=RND(1000)
 30 R1=1E5*RND(1)
 40 R2=1E5*RND(1)
 50 D1=RND(1)
 60 D2=RND(1)
 70 PROC_formprint(I,0,5)
 80 PROC_formprint(R1,2,10)
 90 PROC_formprint(R2,2,10)
100 PROC_formprint(D1,3,7)
110 PROC_formprint(D2,3,7)
120 PRINT
130 NEXT J
140 END
```

Exercise 11.12

The ASCII file should contain the following lines

```
*CAT
NEW
10 PRINT
20 INPUT "WHICH PROGRAM DO YOU WANT TO RUN",P$
30 CHAIN P$
RUN
```

Exercise 11.13

```
 10 REPEAT
 15 CLS
 20 PRINT ''
 30 PRINT "WHICH MATHEMATICAL OPERATION WOULD YOU LIKE?"
 50 PRINT '"1. RECIPROCAL OF N"
 60 PRINT '"2. 10 TO POWER N"
 70 PRINT '"3. FACTORIAL OF N"
 80 PRINT '"4. PI TO N DECIMAL POINTS"
 90 PRINT '"5. N RANDOM NUMBERS"
100 PRINT '"6. END"
120 PRINT '""TYPE THE NUMBER OF YOUR CHOICE"
125 REPEAT
130 A=VAL(GET$)
135 UNTIL A>0 AND A<7
140 CLS
150 IF A=6 THEN END
160 PRINT "TYPE N: ";
170 N=VAL(GET$)
180 ON A GOSUB 200,300,400,500,600 ELSE END
190 PRINT '"PRESS ANY KEY TO CONTINUE":C$=GET$
195 UNTIL FALSE
200 PRINT '"RECIPROCAL OF ";N;" IS ";1/N
210 RETURN
300 PRINT '"10 ^ ";N;" IS ";10^N
310 RETURN
400 PROD=1
410 FOR J=1 TO N
420 PROD=PROD*J
430 NEXT J
440 PRINT '"FACTORIAL ";N;" IS ";PROD
450 RETURN
500 PW=1
510 FOR J=1 TO N: PW=PW*10: NEXT J
520 P=INT(PI*PW)/PW
530 PRINT '"PI TO ";N;" DECIMAL POINTS IS ";P
540 RETURN
600 PRINT 'N;" RANDOM NUMBERS:"'
610 FOR J=1 TO N: PRINT RND(1): NEXT J
620 RETURN
```

Appendix O

THE SUPPLEMENTARY DISC AND TAPE

A supplementary disc or tape is available to accompany this book. Each contains all the example programs and the longer answer programs. Example programs are stored on the tape as the example number with a prefix EX, and answer programs prefixed by A, so Example 1.1 would be EX1.1 and the solution to Exercise 1.1 would be A1.1.

On the disc, because of the limitation of 31 files in the disc directory, programs are stored in sets, by stacking them up in memory and then setting PAGE to the particular program wanted. The example and answer programs are therefore accessed via a two-level menu system, initiated by pressing SHIFT-BREAK. All programs, once accessed, can be freely SAVEd to a new disc, or the whole disc can be copied with *BACKUP.

Purchasers of either disc or tape are entitled to make one working copy either of individual programs or entire tape or disc.

The supplementary disc and tape are distributed by Lostock Software, and each is priced at £6.95 including postage and packing. Please write, enclosing cheque or postal order and specifying whether you require tape, 40 track disc or 80 track disc, to

Lostock Software
13 Cranborne Close
Lostock
Bolton
Lancs BL6 4JG

Please apply directly to Lostock Software, not to Addison-Wesley Publishers Ltd.

INDEX

@% variable, 27, 199, 204, 232–235
  effect on STR$, 48
ABS function, 56, 228
*ACCESS command, 120, 255
ACS function, 57
address, memory, 152
ADVAL function, 189
ampersand, for input in hex, 152
AND operator, 58, 65
apostrophe, 45, 197
appending to a file, 133
arithmetic operations, 7
arrays, 17–21
  declaration, 18
  string, 20, 50–52
  subscript, 18
ASC function, 49
ASCII codes, 48, 237–238
ASCII files, 125, 143–150
  capturing program output in, 147
  file commands, 122
ASN function, 57
assembler, 179–181
assignment statements, 5–8
  for strings, 41
ATN function, 57
auto-repeat, 220

*BACKUP command, 116
BASIC I, 34, 132, 230–231
BASIC II, 132, 230–231
BASIC stack, 155–156
baud rate, setting, 220, 244
BELL, 222–223
BGET# command, 138, 140, 143
binary files, 125, 126–127
binary notation, 151
binary operator, 41
bits, 151
Booleans, 62
  bitwise action of operators, 67
  expressions, 63
  operators, 65
  with IF...THEN, 67
!BOOT files, 123, 145, 210
BPUT# command, 138, 143
branching, 8
BREAK key, 214
bubble sort, 53
buffers
  flushing, 191
  for data files, 127, 154
  input, 154
*BUILD command, 124, 122, 143
*BYE command, 253
bytes, 152

CALL command, 181
Can't extend problem, 134–135
capturing program output, 147
*CAT catalogue command, 111, 256
catalogue structure, 263–265
CHAIN command, 177
channel numbers, 128
channel variable, 128
characters, user-defined, 90–92, 154
CHR$ function, 49, 83, 207
circles
  plotting, 79–80
  solid, 82
CLEAR command, 227
CLG command, 74, 89

CLOSE# command, 122, 129
CLS command, 74
COLOUR command, 72
colour in text and graphics, 70, 72, 206–210
  changing, 89–90
comma, 40, 197, 203
command line interpreter (CLI), 174
*COMPACT command, 120
complete file specification, 112
compress line on input, 14
conditional tests, 63–65
*COPY command, 117
COS function, 57, 231
COUNT function, 202, 228, 236
CTRL-BREAK, 104
  filing system selection, 248
CTRL codes, 224–225
cursor, switching off, 93
cursor keys, 215
  disabling, 190

data files, 125, 127–143
  closing, 129
  format, 139
  opening, 128, 130, 132
  program to read or write, 140–143
  random access, 135–138
  single-byte access, 138–139
DATA statements, 42, 226
DEF PROC statement, 30
default filing system, 108
DEG function, 57
*DELETE command, 118
*DESTROY command, 118
DIMension, 18, 172, 183, 226
*DIR command, 110, 256
directories, 109
*DISC command, 109
disc filing system (DFS), 2, 108–124
  memory usage, 154
  organisation, 119
DISCZAP program, 262–265
*DISK command, 109
DIV operator, 58
double-height text, 208, 227
double-sided disc drives, 110
DRAW command, 74–75
*DRIVE command, 111, 251
*DUMP command, 123, 143–144

Econet, 247–261
  level 1 commands, 250
  level 2 commands, 254
  level 1 error messages, 252
  level 2 error messages, 259
  network filing system (NFS), 109
  network utilities, 251
  printing from, 245
ellipse, 80
*ENABLE command, 116
ENDPROC, 30
EOF# function, 131, 236
EOR operator, 58, 65
equal sign =, 6, 10
EQU pseudo-operations, 180, 231
ERL function, 104, 236
ERR function, 104, 236
errors
  codes, 103, 241–242
  execution, 99
  semantic, 97, 98
  syntax, 95, 96
ESCAPE key, disabling, 104, 190
EVAL function, 57
*EXEC command, 122, 144–146
execution address, 185
EXP function, 57
EXT# function, 133, 236

FALSE pseudo-variable, 63, 236
fields, in a record, 135
file server, 2, 247
file types, 125–127
files
  backup and transfer, 116–117
  commands, table of, 116
  disc (DFS), 2
  Econet, 2
  filing systems, 1, 106
  opening and closing, 128–133
  protection, 120
  removal, 118
  reserved characters, 109
  single-byte access, 138–140
flashing colours, 73
floppy discs, 106–108
flow charts, 22
flushing buffers, 191, 220
FOR...NEXT loop, 10
*FORM command, 113
formatting a disc, 113
formatting numerical output, 202–206, 232
free memory, 157

function keys, 215–217, 223
  use from within a program, 175–176
functions
  mathematical, 56–58
  user-defined, 25
*FX commands, 219–223
  from machine code, 184
  *FX 3, 244
  *FX 4, 190, 217
  *FX 5, 244
  *FX 6, 245
  *FX 8, 244
  *FX 15, 191
  *FX 20, 92
  *FX 138, 176
  *FX 229, 190

GCOL command, 73, 74
GET and GET$ functions, 189
good programming style, 25
GOSUB statement, 32
GOTO statement, 8, 25
graphics
  modes, 69–70
  origin, 85
  window, 88–89

*HELP, 121
hexadecimal notation, 151–152
hierarchical directory, 252
HIMEM pseudo-variable, 155–156, 236

*I AM command, 249, 253
IF...THEN conditional statement, 8
IF...THEN...ELSE...structure, 14
*INFO command, 119
indexed addressing, 161
indirection operators, 159–162
INKEY function, 190–193, 221, 227
  codes, 239–240
INKEY$ function, 190
INPUT command, 40, 41, 189, 194, 230
  LINE with, 41
  suppression of prompt, 40
INPUT# command, 130
INSTR function, 46
  bug in BASIC I, 46, 229
interactive use, 188–193
interlace, turning off, 93, 217
INT function, 56
inverse printing, 209–210
iteration, 23

keyboard, 214–215
*KEY command, 216

labels, in assembly, 182
layout of numbers, 202–203, 232
layout of text, 197–202
LEFT$ function, 44
LEN function, 43
*LIB command, 256
LINE, 41
*LIST command, 123, 143
LISTO command, 16
LN function, 57, 231
*LOAD command, 125–127, 186
LOAD command, 110, 257
loading more than one program, 157–158
LOCAL statement, 28, 31
LOG function, 57, 231
logical colours, 89
logical expressions, 10, 62–68
LOMEM pseudo-variable, 155–156, 236

machine code
  allocating space for, 183
  loading into memory, 147
  use, 178–186
Machine Operating System (MOS), 1, 215
memory access, 159–162
memory organization and usage, 153–155
MEMZAP program, 162–168
menus, 188, 210–213
merging programs, 146
MID$ function, 44
MODE command, 69
  properties of, 70
MODE 7 display, 206–209
MOD operator, 58
MOVE command, 74–75

*NET command, 247
*NOTIFY command, 258
NOT operator, 15, 65

ON ERROR, 102, 129, 229
  limitations, 105
ON...GOSUB statement, 32, 228
ON...GOTO statement, 32, 228
OPENIN function, 130, 132, 230
OPENOUT function, 128–129, 132
OPENUP function, 132–133, 230

operating system commands, 215
  issuing from within a program, 174–175
  from machine code, 183–185
operators, order of priority, 7, 66–67
*OPT command, 119, 123, 145
OPT pseudo–operation, 180, 182
ordering, alphanumeric, 42
origin of coordinates, 71, 85
OR operator, 58, 65
OSBYTE, 184, 219
OSCLI command, 174, 184, 230
OSHWM, 155
OSWORD, 262
OSWRCH, 184

paged mode, 12
pages, memory, 153
PAGE pseudo–variable, 92, 155–157, 236
parabola program, 79
parameter
  actual, 26
  formal. 26, 31
  local, 28
*PASS command, 255
password, 253
PEEK, 159
PI function, 57, 236
pixels, 70–72
PLOT command, 81–83
plotting curves, 77–80
plotting with straight lines, 75–77
pointers
  for files, 133
  memory, 155–157
POKE, 159
pontoon program, 60–62
POS, 85, 199–200, 202, 236
precedence, for operators, 7
preserving variables during chaining, 177–178
PRINT command, 231
  @% for, 232–235
  apostrophe in, 45
PRINT# command, 129
printers, 243–246
  output to, 202
  servers, 247
programs
  ASCII file, to read or write, 148–150, 278–279
  bubble sort, 53–54
  circle, 79–80
  data file, to read or write, 140–142
  DISCZAP, 162, 262–265
  for formatting numbers, 205
  inverse printing, 209–210
  MEMZAP, 162–168
  menu, 123, 211–212
  message in border of stars, 51–52
  parabola, 79
  pontoon game, 60–62
  random access textfiles, 137–138
  sales graph, 75–77
  sine wave, 78
  student mark list, 54–55
  validation, 196–197
  wrap round of text, 52–53
protection against LISTing a program, 170
pseudo–operations, 180
pseudo–variables, 236
PTR# function, 133, 236

RAD function, 57
random access data files, 135–138
random numbers, 58–62
  statistical test of, 60
records, 135
recovering a deleted program, 169
recursion, 34–38
relational operators, 10, 63
  with strings, 65
relocating a BASIC program, 158–159
*RENAME command, 120
RENUMBER command, 227
REPEAT...UNTIL statements, 13
REPORT command, 105
resident integer variables, 5, 86, 173–174
resolution, in graphics, 70
RESTORE command, 26
RIGHT$ function, 44
RND function, 59
rules of precedence, 7
*RUN command, 186, 215

sales graph program, 75–77
SAVE command, 110
*SAVE command, 125–127, 134, 185
scaling, for graphics, 77
screen coordinates, 71
screen editing, 215
semantic errors, 97, 98

semi-colon, 40, 86, 197, 203
sequential files, 140
SGN function, 56
SHIFT enable, 2
sine wave program, 78
SIN function, 57, 231
soft-sectored, 113
sorting, 42, 53
SOUND command, 217–218
SPC, 199–200
spiral, 80
*SPOOL command, 122, 145–147, 228
SQR function, 57
statements
  assignment, 6
  execution, 8
  on the same line, 10
STEP, in FOR...NEXT loops, 11
STOP command, 101
storage of BASIC programs, 168–169
STR$ function, 47–48, 231
  tilde with, 48
  @% with, 48
STRING$ function, 21, 47
strings
  arrays, 20, 50–52
  concatenation, 41
  manipulation functions, 43–50
  reassignment, 21
string variables, 39
  comparison of, 42
  INPUT for, 41
  READ for, 41
structure charts, 22
subroutines, 32
supplementary disc and tape, 284
syntax errors, 95, 96

TAB, 199–202, 203
TAB(X,Y), 201
TAN function, 57
text characters, 70
text layout, 197–202
text window, 86–88
tilde, for output in hex, 152
  with STR$, 48
TIME pseudo-variable, 13, 236
*TITLE command, 121
tokens, 168
top-down design, 22
TOP function, 155–156, 236
track, 107
track density, 113
TRACE command, 100
triangle filling, 82
TRUE pseudo-variable, 63, 236
turnkey system, 123, 210
*TV command, 217
*TYPE command, 122, 143

unary operators, 65
underline character, 5
UNTIL statement, 13
user-defined characters, 90–92
  extra memory for, 92
user-friendly programming, 187–188
USR functions, 181

VAL function, 47–48, 194
validation, 193–197
  numbers, 194–197
  strings, 193–194
variables
  and assignment statements, 5
  common, 28, 30, 31
  control, 100
  global, 28
  integer, 5
  method of storage, 170
  numerical, 5
  preserving during chaining, 177–178
  resident integer, 5, 86, 173–174
VDU commands, 49–50, 83–93, 207, 218, 224–225
  display space, 155
  from machine code, 184
  non-graphics, 84–85, 218
  semi-colons with, 86
  VDU 5, 84, 89
  VDU 23, 90–93
  VDU 24, 88–89
  VDU 28, 87
  VDU 29, 85
*VERIFY command, 114
verifying a disc, 113
version number, of OS, 219
VPOS function, 85, 236

WHILE, 16
WIDTH command, 202
wildcards, 115
windows, 86–89
word processor, for editing programs, 146

The Super-User's BBC Micro Book

The BBC microcomputer is in many ways outstanding for a small microcomputer in the wide range of facilities built into the basic machine and in the power and range of its language and operating system. Indeed, BBC BASIC is so much more extensive and better structured than the versions of BASIC found on most other small microcomputers that it almost needs to be considered as a separate language, of which standard BASIC is but a subset.

The aim of this book is to introduce readers to this treasure house of possibilities. It covers a range of advanced aspects including structured features of BBC BASIC, disc file handling, advanced programming, strings and mathematical functions. Extensive appendices provide excellent reference material on a wide variety of topics, including the Econet file handling system.

Written in a clear self-tutorial style with many examples and exercises, the book should appeal to all keen users of the BBC Micro.

ADDISON-WESLEY PUBLISHING COMPANY

ISBN 0 201 15435 8